Degree in Engineering in Telecommunications Technologies

Bachelor's final project

Scaling LLM Inference on a modern GPU Cluster

Author
Laura González Morán

Supervised by Atilano Ramiro Fernández-Pacheco Sánchez-Migallón -
Universidad Pontificia Comillas ICAI

Madrid
August 2025

Laura González Morán, declara bajo su responsabilidad, que el Proyecto con título **Scaling LLM Inference on a modern GPU Cluster** presentado en la ETS de Ingeniería (ICAI) de la Universidad Pontificia Comillas en el curso académico 2024/25 es de su autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Laura González Morán Fecha: 22 / 08 / 2025

Autoriza la entrega:

El Director del Proyecto

Atilano Ramiro Fernández-Pacheco Sánchez-Migallón

Fdo.: ..... Fecha: ..... / ..... / .....

71216314B ATILANO RAMIRO FERNÁNDEZ-PACHECO

Digitally signed by 71216314B ATILANO RAMIRO FERNÁNDEZ-PACHECO
Date: 2025.08.25 22:33:59 +02'00'

Firmado por GONZALEZ MORAN, LAURA (FIRMA) el día 26/08/2025 con un certificado emitido por AC DNIE 006

Degree in Engineering in Telecommunications Technologies

Bachelor's final project

Scaling LLM Inference on a modern GPU Cluster

Author
Laura González Morán

Supervised by
Atilano Ramiro Fernández-Pacheco Sánchez-Migallón - Universidad Pontificia Comillas ICAI

Madrid
August 2025

Acknowledgements

I would first like to express my gratitude to the Systems Platform Research Group at the University of Illinois Urbana-Champaign, for generously welcoming me into their team and providing access to their computing infrastructure.

I am deeply thankful to my family, whose support has been a constant source of strength for me, and whose unwavering encouragement and love have accompanied me throughout my whole education.

Finally, I would like to sincerely thank Atilano, both for supervising this bachelor's thesis and for being an inspiring professor and mentor. His guidance and dedication have been invaluable, and I feel very fortunate to have learned under his direction.

Scaling LLM Inference on a Modern GPU Cluster

Author: González Morán, Laura

Director: Fernández-Pacheco Sánchez-Migallón, Atilano Ramiro

Collaborating Entity: ICAI – Universidad Pontificia Comillas

Abstract

This Bachelor's Thesis conducts a multivariable analysis of large-language-model (LLM) inference performance at scale, with the goal of identifying the primary factors driving resource inefficiency and their economic implications for infrastructure providers. To enable this analysis, an automated benchmarking pipeline was developed to deploy open-source LLMs on an H100 GPU cluster, generate realistic user workloads through controlled request streams, and capture fine-grained hardware telemetry in real time. By systematically varying key parameters such as concurrency, model size, sequence length, and tensor parallelism, the study characterises how different workload profiles affect throughput, latency, and resource saturation. The empirical results are used to derive role-aware operating recommendations tailored to developers, system architects, infrastructure providers, and hardware manufacturers, offering practical guidance for improving the efficiency, scalability, and economic viability of LLM inference systems.

Keywords: Large Language Models, Inference, Benchmarking, KV-Cache, Scalability, vLLM, AI Infrastructure.

1. Introduction

Generative AI inference platforms are nowadays used simultaneously by very large user populations, and that volume keeps growing steadily [1]. Sustaining such a user request rate requires a data center infrastructure equipped with state-of-the-art Graphics Processing Units (GPUs) [2]. However, their acquisition and operation involve multibillion-dollar investments, so optimising these resources is essential to contain costs [3, 4]. As these workloads grow in complexity and volume, they place increasing strain on the underlying hardware, exposing bottlenecks that limit throughput and degrade efficiency. Even small inefficiencies can accumulate at scale, leading to disproportionate resource consumption and escalating operational costs.

The economic impact has already become visible in industry: even companies generating billions in revenue from AI services have reported difficulty achieving profitability due to the overwhelming cost of operating large-scale inference [5, 3, 4]. In other words, without fundamental efficiency improvements, the business models behind these platforms become unsustainable.

2. Project Definition

The purpose of this Final Degree Project is to produce measurable insight into how large language model inference scales in practice and to convert that insight into actionable guidance for those who build and operate these systems.

To that end, the work conducts a multivariable analysis of large-scale LLM inference to identify the factors that most drive resource inefficiency and thereby worsen the cost structure for infrastructure providers. It then designs and implements a reproducible benchmarking pipeline that deploys open-source language models on an H100 cluster, generates realistic streams of user requests, and captures fine-grained hardware telemetry in real time so that performance can be characterised across workloads. Finally, it distils the empirical findings into role-aware recommendations that improve the efficiency of operating LLM inference services.

These aims are summarised below:

- **Conduct a multivariable analysis of large-scale LLM inference:** Quantify how performance and resource efficiency change as key workload and system factors vary, and identify the main drivers of underutilisation that deteriorate providers' cost structure.
- **Design a reproducible benchmarking pipeline:** Build an automated and repeatable pipeline that deploys open-source LLMs on an H100 cluster, generates synthetic user workloads, and records fine-grained hardware metrics at sub-second resolution to produce the datasets required for the analysis.
- **Deliver role-aware operating guidance grounded in empirical results:** Translate measured behaviour into data-driven recommendations for developers, system architects, infrastructure providers, and hardware manufacturers so each can apply the findings to improve efficiency across the AI ecosystem.

3. System description

The implemented system has been designed as a modular benchmarking pipeline, where each subsystem fulfills a precise responsibility while remaining independent from the others. The architecture was structured to guarantee automation, reproducibility, and extensibility, allowing the entire benchmark to be executed with a single command or each component to be invoked individually for debugging and validation.

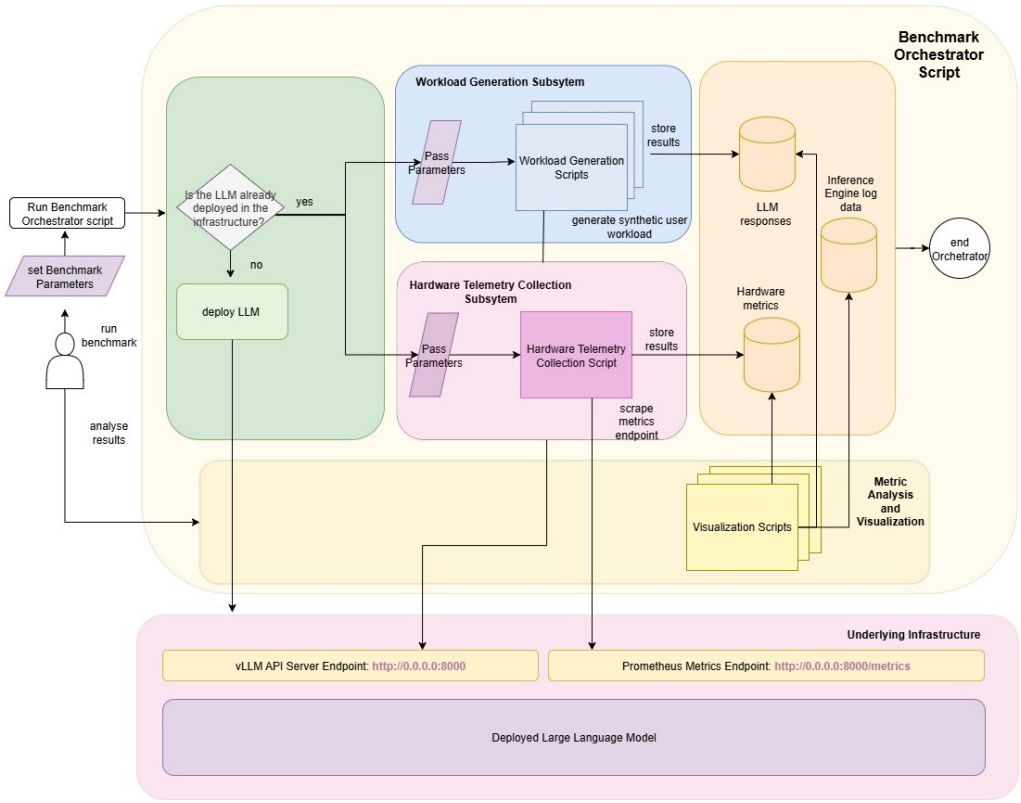


Figure 1: Pipeline Orchestration Schema

The platform is divided into five main functional blocks. The first is *Benchmark Selection*, which defines the open-source models and workload configurations used to reflect realistic large-scale inference scenarios. The second block, *Automated Pipeline Orchestration*, coordinates the execution of all subsystems through shell scripts and configuration files, ensuring the correct sequence of operations without manual intervention. The third block, *User Workload Generation*, produces realistic streams of concurrent requests that vary in prompt length, sequence length, and concurrency, closely mirroring production user behaviour. The fourth block,

Hardware Telemetry Collection, captures fine-grained GPU and system-level metrics in real time using Prometheus, storing the results for later analysis. Finally, the block of *Metric Analysis and Visualization* processes the recorded data to derive insights on throughput, latency, and memory utilisation, highlighting scaling behaviours and bottlenecks.

Overall, this modular design makes the pipeline flexible and transparent. Each component can evolve independently, while their integration provides a comprehensive end-to-end benchmarking system that is both reproducible and efficient.

4. Results

The experimental results gathered in this project thanks to the development of an automated benchmarking system, allowed for a multivariable analysis of LLM inference performance across a wide range of model architectures and workload configurations. This analysis consistently identified GPU memory saturation (particularly coming from KV-cache accumulation) as the primary bottleneck that limits throughput and drives up latency under load. While compute resources were often underutilised, it was memory exhaustion that most frequently triggered system-wide slowdowns and rendered the infrastructure economically inefficient.

This insight proved essential for understanding when and why inference workloads fail to scale, and how such failures degrade the cost structure for infrastructure providers. Armed with this empirical evidence, the final phase of the project translated these findings into a set of concrete, role-aware recommendations. These recommendations target the four key actors involved in the deployment of large-scale AI systems: developers, system architects, infrastructure providers, and hardware manufacturers.

Each recommendation is tailored to the decisions and tradeoffs that stakeholders must routinely navigate. The following list provides a brief summary of the final recommendations presented in the *Applied Recommendations Informed by Empirical Findings* section:

- **Developers** are guided to select models with *memory-efficient attention mechanisms*, as these significantly reduce the risk of KV-cache saturation during inference.
- **System Architects** are advised to *specialise their clusters by workload type*, avoiding general-purpose architectures that often result in unpredictable saturation behaviour and inefficient scaling.
- **Infrastructure Providers** are encouraged to *offer purpose-built infrastructure profiles* aligned with specific workload demands, and to maintain optimal

operating conditions through *real-time telemetry monitoring and adaptive scaling*.

- **Hardware Manufacturers** are prompted to *prioritise innovation in memory design*, in line with recent industry trends toward *memory pooling*, *hierarchical KV-cache architectures*, and *disaggregated handling of prefill and decode phases*.

Together, these recommendations form a unified, evidence-based strategy for improving the efficiency, scalability, and economic viability of LLM inference systems across the entire AI infrastructure stack.

5. Conclusions

Once the development of this project has concluded, and based on the results obtained, it can be confirmed that all proposed objectives have been successfully achieved. A reproducible benchmarking pipeline was designed and implemented to deploy open-source large language models on an H100 GPU cluster, generate realistic user workloads, and collect high-resolution hardware telemetry. This system enabled a multivariable analysis of inference performance across different models and workload conditions.

The results revealed that GPU memory saturation (especially from KV-cache accumulation) was the most prominent bottleneck limiting throughput and increasing latency under load. This finding provides a deeper understanding of the operational limits of current infrastructure and their implications for scalability and cost-efficiency.

Finally, the insights derived from these benchmarks were used to develop targeted recommendations for each stakeholder role across the AI infrastructure stack. These recommendations offer actionable guidance for improving system-level efficiency, supporting better technical and economic outcomes for future LLM inference deployments.

6. References

[1] Emma Roth. *ChatGPT's weekly users have doubled in less than a year*. https://www.theverge.com/2024/8/29/24231685/openai-chatgpt-200-million-weekly-users. The Verge, 29 Aug 2024. Aug. 2024. (Visited on 08/22/2025).

[2] Brian Caulfield. *What's the Difference Between a CPU and a GPU?* https://blogs.nvidia.com/blog/whats-the-difference-between-a-cpu-and-a-gpu/. NVIDIA Blog. Original post Dec 2009; periodically updated. Dec. 2009. (Visited on 08/22/2025).

[3] Kenrick Cai. *Alphabet reaffirms $75 billion spending plan in 2025 despite tariff turmoil*. https://www.reuters.com/technology/alphabet-ceo-reaffirms-planned-75-billion-capital-spending-2025-2025-04-09/. Reuters, 10 Apr 2025. Apr. 2025. (Visited on 08/22/2025).

[4] Reuters. *Microsoft to spend record $30 billion this quarter as AI investments pay off*. https://www.reuters.com/business/microsoft-spend-record-30-billion-this-quarter-ai-investments-pay-off-2025-07-30/. Reuters, 30 Jul 2025. July 2025. (Visited on 08/22/2025).

[5] Reuters. *OpenAI hits $12 billion in annualized revenue, The Information reports*. https://www.reuters.com/business/openai-hits-12-billion-annualized-revenue-information-reports-2025-07-31/. Reuters, 31 Jul 2025. July 2025. (Visited on 08/22/2025).

Escalado de la inferencia de LLMs en un clúster moderno de GPUs

Autor: González Morán, Laura

Director: Fernández-Pacheco Sánchez-Migallón, Atilano Ramiro

Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

Resumen

Este Trabajo de Fin de Grado realiza un análisis multivariable del rendimiento de la inferencia de modelos de lenguaje de gran escala (LLMs), con el objetivo de identificar los principales factores que generan ineficiencia en el uso de recursos y sus implicaciones económicas para los proveedores de infraestructura. Para llevar a cabo este análisis, se desarrolló una canalización de pruebas automatizada que permite desplegar LLMs de código abierto sobre un clúster de GPUs H100, generar cargas de trabajo realistas mediante flujos de peticiones controladas, y capturar telemetría de hardware de alta resolución en tiempo real. Variando sistemáticamente parámetros clave como la concurrencia, el tamaño del modelo, la longitud de la secuencia y el nivel de paralelismo tensorial, el estudio caracteriza cómo diferentes perfiles de carga afectan al rendimiento en términos de throughput, latencia y saturación de recursos. Los resultados empíricos se utilizan para derivar recomendaciones operativas adaptadas a cada uno de los principales roles implicados en la infraestructura de IA —desarrolladores, arquitectos de sistemas, proveedores de infraestructura y fabricantes de hardware— ofreciendo así una guía práctica para mejorar la eficiencia, escalabilidad y viabilidad económica de los sistemas de inferencia con LLMs.

Palabras clave: Modelos de Lenguaje de Gran Escala, Inferencia, Benchmarking, KV-Cache, Escalabilidad, vLLM, Infraestructura de IA.

1. Introducción

Las plataformas de inferencia de IA generativa son utilizadas hoy en día de forma simultánea por poblaciones de usuarios muy numerosas, y ese volumen continúa creciendo de manera constante [1]. Sostener este ritmo de peticiones requiere una infraestructura de centros de datos equipada con Unidades de Procesamiento Gráfico (GPUs) de última generación [2]. Sin embargo, su adquisición y operación

implican inversiones de varios miles de millones de dólares, por lo que optimizar estos recursos resulta esencial para contener los costes [3, 4].

A medida que estas cargas de trabajo crecen en complejidad y volumen, ejercen una presión cada vez mayor sobre el hardware subyacente, exponiendo cuellos de botella que limitan el rendimiento y degradan la eficiencia. Incluso pequeñas ineficiencias pueden acumularse a gran escala, provocando un consumo desproporcionado de recursos y un aumento significativo de los costes operativos.

El impacto económico ya se ha hecho visible en la industria: incluso empresas que generan miles de millones en ingresos por servicios de IA han reportado dificultades para alcanzar la rentabilidad debido al elevado coste de operar inferencia a gran escala [5, 3, 4]. En otras palabras, sin mejoras fundamentales en eficiencia, los modelos de negocio detrás de estas plataformas se vuelven insostenibles.

2. Definición del Proyecto

El propósito de este Trabajo de Fin de Grado es ofrecer una visión cuantitativa sobre cómo escala en la práctica la inferencia de grandes modelos de lenguaje (LLMs), y transformar ese conocimiento en recomendaciones prácticas para quienes diseñan y operan estos sistemas.

Con ese fin, el trabajo realiza un análisis multivariable de la inferencia de LLMs a gran escala, con el objetivo de identificar los factores que más contribuyen a la ineficiencia en el uso de recursos y que, en consecuencia, deterioran la estructura de costes de los proveedores de infraestructura. A continuación, diseña e implementa una infraestructura de benchmarking reproducible que despliega modelos de lenguaje de código abierto sobre un clúster de GPUs H100, genera flujos realistas de peticiones de usuarios, y recoge telemetría de hardware de alta resolución en tiempo real para caracterizar el rendimiento bajo diferentes cargas de trabajo. Por último, destila los hallazgos empíricos en recomendaciones adaptadas a los distintos roles del ecosistema, que permiten mejorar la eficiencia operativa de los servicios de inferencia.

Estos objetivos se resumen a continuación:

- **Realizar un análisis multivariable de la inferencia de LLMs a gran escala:** Cuantificar cómo cambian el rendimiento y la eficiencia en el uso de recursos al variar distintos factores del sistema y de la carga de trabajo, e identificar los principales causantes de infrautilización que afectan negativamente a la estructura de costes de los proveedores.
- **Diseñar una infraestructura de benchmarking reproducible:** Construir una infraestructura automatizada y repetible que despliegue modelos

LLMs de código abierto en un clúster H100, genere cargas sintéticas de trabajo de usuario, y registre métricas de hardware a resolución sub-segundo para generar los datos necesarios para el análisis multivariable.

- **Ofrecer recomendaciones operativas basadas en datos empíricos y adaptadas por rol:** Traducir el comportamiento observado en recomendaciones prácticas para desarrolladores, arquitectos de sistemas, proveedores de infraestructura y fabricantes de hardware, de forma que cada uno pueda aplicar los hallazgos para mejorar la eficiencia en su capa del ecosistema de IA.

3. Descripción del Sistema

El sistema implementado ha sido diseñado como una infraestructura de benchmarking modular, donde cada subsistema cumple una responsabilidad precisa manteniendo su independencia respecto a los demás. La arquitectura se ha estructurado para garantizar automatización, reproducibilidad y extensibilidad, permitiendo ejecutar todo el benchmark con un solo comando o invocar cada componente de forma individual para tareas de depuración y validación.

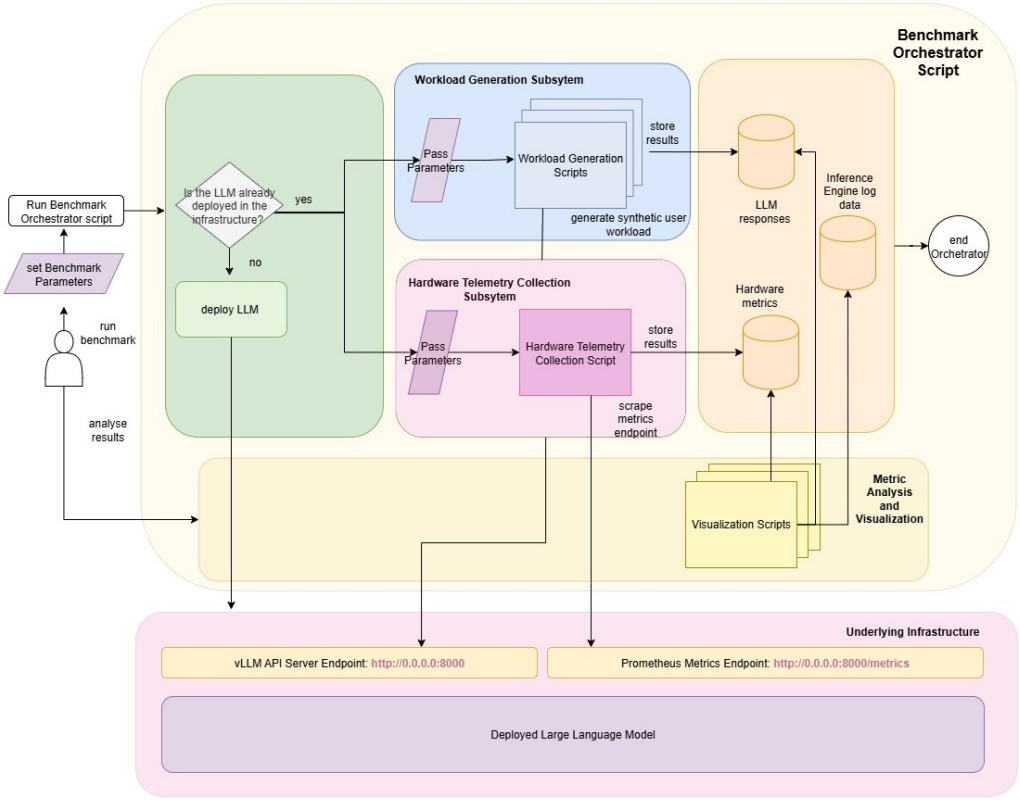


Figure 2: Esquema de Orquestación del Pipeline

La plataforma se divide en cinco bloques funcionales principales. El primero es *Selección de Benchmark*, que define los modelos de código abierto y las configuraciones de carga de trabajo utilizadas para reflejar escenarios realistas de inferencia a gran escala. El segundo bloque, *Orquestación Automatizada del Pipeline*, coordina la ejecución de todos los subsistemas mediante scripts de shell y archivos de configuración, garantizando la secuencia correcta de operaciones sin intervención manual. El tercer bloque, *Generación de Carga de Trabajo de Usuario*, produce flujos realistas de peticiones concurrentes que varían en longitud del prompt, longitud de secuencia y concurrencia, simulando fielmente el comportamiento de usuarios en producción. El cuarto bloque, *Captura de Telemetría de Hardware*, recoge métricas detalladas de GPU y del sistema en tiempo real utilizando Prometheus, y almacena los resultados para su posterior análisis. Finalmente, el bloque de *Análisis y Visualización de Métricas* procesa los datos registrados para obtener conclusiones sobre rendimiento, latencia y uso de memoria, destacando patrones de escalado y cuellos de botella.

En conjunto, este diseño modular hace que el pipeline sea flexible y transparente.

Cada componente puede evolucionar de forma independiente, mientras que su integración proporciona un sistema de benchmarking de extremo a extremo que es tanto reproducible como eficiente.

4. Resultados

Los resultados experimentales obtenidos en este proyecto, gracias al desarrollo de un sistema automatizado de benchmarking, permitieron llevar a cabo un análisis multivariable del rendimiento de inferencia de modelos de lenguaje a gran escala (LLM) en una amplia gama de arquitecturas de modelo y configuraciones de carga de trabajo. Este análisis identificó de forma consistente la saturación de memoria GPU (en particular, la acumulación de KV-cache) como el principal cuello de botella que limita el rendimiento y eleva la latencia bajo carga. Aunque los recursos de cómputo permanecían frecuentemente infrautilizados, era la falta de memoria la que más habitualmente provocaba ralentizaciones generalizadas y volvía ineficiente la infraestructura desde el punto de vista económico.

Este hallazgo resultó esencial para entender cuándo y por qué las cargas de inferencia dejan de escalar, y cómo estos fallos degradan la estructura de costes para los proveedores de infraestructura. Con esta evidencia empírica, la fase final del proyecto consistió en traducir estos hallazgos en un conjunto de recomendaciones concretas y adaptadas a cada rol. Estas recomendaciones están dirigidas a los cuatro actores clave implicados en el despliegue de sistemas de inteligencia artificial a gran escala: desarrolladores, arquitectos de sistemas, proveedores de infraestructura y fabricantes de hardware.

Cada recomendación está adaptada a las decisiones y compromisos que estos actores deben afrontar de forma rutinaria. La siguiente lista proporciona un resumen de las recomendaciones finales presentadas en la sección *Recomendaciones Aplicadas Informadas por Resultados Empíricos*:

- **Desarrolladores:** se les orienta a seleccionar modelos con *mecanismos de atención eficientes en memoria*, ya que estos reducen significativamente el riesgo de saturación de KV-cache durante la inferencia.
- **Arquitectos de Sistemas:** se les recomienda *especializar sus clústeres según el tipo de carga de trabajo*, evitando arquitecturas generalistas que suelen producir patrones de saturación impredecibles y un escalado ineficiente.
- **Proveedores de Infraestructura:** se les anima a *ofrecer perfiles de infraestructura especializados* adaptados a las necesidades de cargas de trabajo concretas, y a mantener condiciones óptimas de operación mediante

monitorización de telemetría en tiempo real y escalado adaptativo.

- **Fabricantes de Hardware:** se les insta a *priorizar la innovación en el diseño de memoria*, en línea con las tendencias actuales del sector hacia *agrupamiento de memoria (memory pooling)*, *arquitecturas jerárquicas de KV-cache* y *gestión desagregada de las fases de prefill y decode.*

En conjunto, estas recomendaciones constituyen una estrategia unificada y basada en evidencia para mejorar la eficiencia, escalabilidad y viabilidad económica de los sistemas de inferencia de LLM en toda la pila de infraestructura de IA.

5. Conclusiones

Una vez finalizado el desarrollo de este proyecto, y con base en los resultados obtenidos, se puede confirmar que todos los objetivos propuestos han sido alcanzados con éxito. Se diseñó e implementó una pipeline de benchmarking reproducible para desplegar modelos de lenguaje abiertos en un clúster de GPUs H100, generar cargas de trabajo realistas de usuarios y recopilar telemetría de hardware con alta resolución. Este sistema permitió realizar un análisis multivariable del rendimiento de inferencia bajo distintas condiciones de modelo y carga.

Los resultados revelaron que la saturación de memoria GPU (especialmente debido a la acumulación de KV-cache) fue el cuello de botella más destacado, limitando el rendimiento y aumentando la latencia bajo carga. Este hallazgo proporciona una comprensión más profunda de los límites operativos de la infraestructura actual y sus implicaciones en términos de escalabilidad y eficiencia de costes.

Finalmente, los conocimientos extraídos de estos benchmarks fueron utilizados para desarrollar recomendaciones específicas dirigidas a cada uno de los roles dentro de la pila de infraestructura de IA. Estas recomendaciones ofrecen orientación práctica para mejorar la eficiencia a nivel de sistema, facilitando mejores resultados técnicos y económicos en futuros despliegues de inferencia de modelos de lenguaje.

6. Referencias

[1] Emma Roth. *ChatGPT's weekly users have doubled in less than a year.* https://www.theverge.com/2024/8/29/24231685/openai-chatgpt-200-million-weekly-users. The Verge, 29 Aug 2024. Aug. 2024. (Visited on 08/22/2025).

[2] Brian Caulfield. *What's the Difference Between a CPU and a GPU?* https://blogs.nvidia.com/blog/whats-the-difference-between-a-cpu-and-a-gpu/. NVIDIA Blog. Original post Dec 2009; periodically updated. Dec. 2009. (Visited on 08/22/2025).

[3] Kenrick Cai. *Alphabet reaffirms $75 billion spending plan in 2025 despite tariff turmoil*. https://www.reuters.com/technology/alphabet-ceo-reaffirms-planned-75-billion-capital-spending-2025-2025-04-09/. Reuters, 10 Apr 2025. Apr. 2025. (Visited on 08/22/2025).

[4] Reuters. *Microsoft to spend record $30 billion this quarter as AI investments pay off*. https://www.reuters.com/business/microsoft-spend-record-30-billion-this-quarter-ai-investments-pay-off-2025-07-30/. Reuters, 30 Jul 2025. July 2025. (Visited on 08/22/2025).

[5] Reuters. *OpenAI hits $12 billion in annualized revenue, The Information reports*. https://www.reuters.com/business/openai-hits-12-billion-annualized-revenue-information-reports-2025-07-31/. Reuters, 31 Jul 2025. July 2025. (Visited on 08/22/2025).

Contents

1 Introduction **1**

2 Description of Technologies **4**
2.1 NVIDIA H100 NVL . . . 4
2.2 vLLM Inference Engine . . . 5
2.3 Hugging Face . . . 6
2.4 Prometheus . . . 6
2.5 Python . . . 7
2.5.1 Pandas . . . 8
2.5.2 Matplotlib . . . 9

3 State of the Art **10**
3.1 Stages in the Life Cycle of an LLM . . . 10
3.1.1 Distributed Training . . . 11
3.1.2 Production-Grade Inference . . . 14
3.2 Transformer Architecture . . . 15
3.2.1 Why transformers have led the AI boom . . . 16
3.2.2 Architectural breakdown of Transformers . . . 19
3.2.3 The Attention Mechanism . . . 25
3.3 From Theory to Execution: Infrastructure behind LLMs . . . 28
3.3.1 Data Centers . . . 29
3.3.2 GPU Memory and Hardware Constraints . . . 31
3.3.3 Quantization . . . 33
3.4 Scaling for Real-World Inference . . . 34
3.4.1 Inference SLOs . . . 34
3.4.2 Parallelism Strategies . . . 36
3.4.3 The Quadratic Complexity in the Attention Mechanism . . . 40
3.5 KV Caching and the Rise of Memory Bottlenecks . . . 41
3.5.1 KV Caching . . . 41
3.5.2 Memory Optimization Techniques . . . 43

3.5.3 Inference Engines . . . 47

4 Project Definition **49**
4.1 Motivation . . . 49
4.2 Objectives . . . 50
4.3 Methology . . . 51
4.4 Planning . . . 52
4.5 Economic Study . . . 53
4.5.1 Cost Structure of Infrastructure Providers . . . 56
4.5.2 Revenue Structure for Infrastructure Providers . . . 57
4.5.3 Profit Structure for Infrastructure Providers . . . 58

5 Implemented System **61**
5.1 Benchmark Selection . . . 62
5.1.1 Concurrency-Related Benchmarks . . . 64
5.1.2 Sequence Length-Related Benchmarks . . . 68
5.1.3 Intra-node Scaling-Related Benchmarks . . . 72
5.2 Pipeline Orchestration . . . 77
5.2.1 Purpose and Motivation . . . 77
5.2.2 High-Level End-to-End Flow . . . 78
5.2.3 Architectural Decisions . . . 79
5.2.4 Benefits of This Approach . . . 80
5.3 Workload Generation . . . 81
5.3.1 Purpose and Role in the Benchmarking Pipeline . . . 81
5.3.2 Model Access and Request Handling . . . 82
5.3.3 Fundamental Benchmarking Unit: Single Request Flow . . . 83
5.3.4 Parameterization for Flexibility and Reproducibility . . . 84
5.3.5 Benchmark Configuration Parameters . . . 85
5.3.6 Structured Output and Result Logging . . . 87
5.4 Hardware Telemetry Collection . . . 88
5.4.1 Purpose and Role in the Benchmarking Pipeline . . . 89
5.4.2 Data Source and Collection Method . . . 89
5.4.3 Metric Parsing and Storage . . . 92
5.4.4 Output Organization . . . 94
5.4.5 Architectural Choices and Implications . . . 94
5.5 Metric Analysis and Visualization Subsytem . . . 95
5.5.1 Purpose and Role in the Benchmarking Pipeline . . . 95
5.5.2 Input Data Preparation . . . 96
5.5.3 Metric Selection and Filtering . . . 96
5.5.4 Visualization and Output Storage . . . 97
5.5.5 Abstraction Benefits . . . 97

6 Analysis and Interpretation of Results **99**
6.1 Analysis of Concurrency-Driven Performance Scaling 99
6.1.1 Analysis of GPU-Memory Saturation Across Concurrency Levels 99
6.1.2 Analysis of Queue Formation Triggered by KV-Cache Ceiling 105
6.1.3 Analysis of KV-Cache Saturation Effects on Aggregate Throughput 113
6.1.4 Analysis of Peak Throughput under Varying Concurrency . . 121
6.1.5 Analysis of Latency Degradation Under Increasing Concurrency 123
6.1.6 Analysis of Model Size on Concurrency Limits 124
6.2 Analysis of Prompt and Generation Sequence Lengths on Inference Latency 126
6.2.1 Prefill Only 126
6.3 Analysis of Intra-Node Scaling Effects on Inference Performance . . 127
6.4 Applied Recommendations Informed by Empirical Findings 130

7 Conclusions **136**
7.0.1 Achieved Objectives 136
7.0.2 Future Work 137

Bibliography **138**

A Alignment with the Sustainable Development Goals **144**

List of Figures

1 Pipeline Orchestration Schema iv
2 Esquema de Orquestación del Pipeline xi

2.1 NVIDIA H100 NVL 5
2.2 NVIDIA Logo 5
2.3 vLLM Logo 5
2.4 Hugging Face Logo 6
2.5 Promethe Logo 7
2.6 Python Logo 8
2.7 Pandas Logo 9
2.8 Matplotlib Logo 9

3.1 Large Scale Distributed Training 11
3.2 Inference 11
3.3 Learning stages in model training 12
3.4 Autorregressive mechanism in LLM inference 14
3.5 Multi-layer Perceptron 16
3.6 Recurrent Neural Network 17
3.7 Tokenization 20
3.8 Encoder-decoder Transformer 22
3.9 Decoder-only Transformer 23
3.10 Autoregressive decoding 23
3.11 A decoder-only Transformer block 25
3.12 Mathematical expression of the Attention mechanism 26
3.13 Query, Key and Value computation 27
3.14 Multi-Head Attention 28
3.15 Abstraction Layers in a Data Center 29
3.16 Performance comparison of LLM Inference with H100 and B300 done by NVIDIA 32
3.17 Quantization example 33
3.18 Breakdown of Inference latency that impacts SLO targets 34

3.19 Comparison of Parallelism Strategies . . . 37
3.20 Column-wise vs Row-wise Tensor Parallelism . . . 38
3.21 An example of 3D-parallelism with data-parallelism, tensor parallelism, and pipeline parallelism. . . . 40
3.22 Paged Attention . . . 45

4.1 Agile Development Cycle . . . 51
4.2 Infrastructure Performance Profile . . . 53
4.3 Concurrency threshold for KV-cache saturation as a function of model size across different models. . . . 54
4.4 Total Cost Structure for Infrastructure Providers . . . 57
4.5 Profit Structure for Infrastructure Providers . . . 59

5.1 Benchmark Families . . . 64
5.2 Concurrency-Sweep Setup: Increasing Numbers of Simultaneous User Requests Served by the Inference System . . . 65
5.3 Sequence Length Setup: Increasing User Prompt Length Served to the Inference System . . . 69
5.4 Intra-node Setup: Increasing Tensor Parallelism in the Inference System . . . 73
5.5 Pipeline Orchestration Schema . . . 77
5.6 Workload Generation Sub-system Schema . . . 81
5.7 View of the end-to-end flow for a single simulated user request . . . 84
5.8 Hardware Telemetry Collection Sub-system Schema . . . 88
5.9 View of the end-to-end flow for the Metrics Analysis and Visualization subsystem. . . . 96

6.1 GPU KV-cache saturation curve for GPT-2-XL with increasing user requests . . . 100
6.2 GPU KV-cache saturation curve for DeepSeek-LLM-7B with increasing user requests . . . 102
6.3 GPU KV-cache saturation curve for Qwen1.5-14B-Chat with increasing user requests . . . 103
6.4 Admission, Queueing, and Memory Pressure. GPT-2-XL (1,200 concurrent user requests) . . . 106
6.5 Admission, Queueing, and Memory Pressure. Deepseek-7B (600 concurrent user requests) . . . 107
6.6 Admission, Queueing, and Memory Pressure. Qwen1.5-14B-Chat (160 concurrent user requests) . . . 109
6.7 Admission, Queueing, and Memory Pressure. Falcon-40B (160 concurrent user requests) . . . 110

6.8 Throughput over time for GPT-2-XL-1.5B with 1 concurrent request. 114
6.9 Throughput over time for GPT-2-XL-1.5B with 10 concurrent requests. . . . 114
6.10 Throughput over time for GPT-2-XL-1.5B with 50 concurrent requests. . . . 114
6.11 Throughput over time for GPT-2-XL-1.5B with 100 concurrent requests. . . . 114
6.12 Throughput over time for GPT-2-XL-1.5B with 200 concurrent requests. . . . 114
6.13 Throughput over time for GPT-2-XL-1.5B with 300 concurrent requests. . . . 114
6.14 Throughput over time for GPT-2-XL-1.5B with 400 concurrent requests. . . . 114
6.15 Throughput over time for GPT-2-XL-1.5B with 500 concurrent requests. . . . 114
6.16 Throughput over time for GPT-2-XL-1.5B with 600 concurrent requests. . . . 115
6.17 Throughput over time for GPT-2-XL-1.5B with 700 concurrent requests. . . . 115
6.18 Throughput over time for GPT-2-XL-1.5B with 800 concurrent requests. . . . 115
6.19 Throughput over time for GPT-2-XL-1.5B with 900 concurrent requests. . . . 115
6.20 Throughput over time for GPT-2-XL-1.5B with 1000 concurrent requests. . . . 115
6.21 Throughput over time for GPT-2-XL-1.5B with 1100 concurrent requests. . . . 115
6.22 Throughput over time for GPT-2-XL-1.5B with 1200 concurrent requests. . . . 115
6.23 Throughput over time for DeepSeek with 1 concurrent request. . . . 116
6.24 Throughput over time for DeepSeek with 10 concurrent requests. . . 116
6.25 Throughput over time for DeepSeek with 50 concurrent requests. . . 116
6.26 Throughput over time for DeepSeek with 100 concurrent requests. . 116
6.27 Throughput over time for DeepSeek with 200 concurrent requests. . 116
6.28 Throughput over time for DeepSeek with 300 concurrent requests. . 116
6.29 Throughput over time for DeepSeek with 400 concurrent requests. . 116
6.30 Throughput over time for DeepSeek with 500 concurrent requests. . 116
6.31 Throughput over time for DeepSeek with 600 concurrent requests. . 116
6.32 Throughput over time for Qwen-14B with 1 concurrent request. . . 117
6.33 Throughput over time for Qwen-14B 10 concurrent requests. . . . 117

6.34 Throughput over time for Qwen-14B 20 concurrent requests. 117
6.35 Throughput over time for Qwen-14B 30 concurrent requests. 117
6.36 Throughput over time for Qwen-14B 40 concurrent requests. 117
6.37 Throughput over time for Qwen-14B 60 concurrent requests. 117
6.38 Throughput over time for Qwen-14B 70 concurrent requests. 117
6.39 Throughput over time for Qwen-14B 80 concurrent requests. 117
6.40 Throughput over time for Qwen-14B 90 concurrent requests. 118
6.41 Throughput over time for Qwen-14B 100 concurrent requests. . . . 118
6.42 Throughput over time for Qwen-14B 110 concurrent requests. . . . 118
6.43 Throughput over time for Qwen-14B 120 concurrent requests. . . . 118
6.44 Throughput over time for Qwen-14B 130 concurrent requests. . . . 118
6.45 Throughput over time for Qwen-14B 140 concurrent requests. . . . 118
6.46 Throughput over time for Qwen-14B 150 concurrent requests. . . . 118
6.47 Throughput over time for Qwen-14B 160 concurrent requests. . . . 118
6.48 Concurrency threshold for KV-cache saturation as a function of model size across different models. 122
6.49 TTFT vs. concurrency for GPT2-XL-chat. 124
6.50 TTFT vs. concurrency for Deepseek-LLM-7B-chat. 124
6.51 TTFT vs. concurrency for Qwen-1.5-14B. 124
6.52 TTFT vs. concurrency for Falcon40B. 124
6.53 Concurrency threshold for KV-cache saturation as a function of model size 125
6.54 TTFT vs. prompt length for Mistral-7B-128k. 126
6.55 Max TTFT vs. prefill input length (Qwen, prefill-only benchmark). 126
6.56 Throughput vs. KV-cache utilisation (Tensor Parallel = 1, Llama-3.1-70B-FP8, 50req) 128
6.57 Throughput vs. KV-cache utilisation (Tensor Parallel = 2, Llama-3.1-70B-FP8, 50req) 128
6.58 mThroughput vs. KV-cache utilisation (Tensor Parallel = 1, Llama-3.1-70B-FP8, 100req) 128
6.59 Throughput vs. KV-cache utilisation (Tensor Parallel = 2, Llama-3.1-70B-FP8, 100req) 128
6.60 Throughput vs. KV-cache utilisation (Tensor Parallel = 1, Llama-3.1-70B-FP8, 150req) 128
6.61 Throughput vs. KV-cache utilisation (Tensor Parallel = 2, Llama-3.1-70B-FP8, 150req) 128
6.62 Throughput vs. KV-cache utilisation (Tensor Parallel = 1, Llama-3.1-70B-FP8, 200req) 129
6.63 Throughput vs. KV-cache utilisation (Tensor Parallel = 2, Llama-3.1-70B-FP8, 200req) 129

6.64 Throughput vs. KV-cache utilisation (Tensor Parallel = 1, Llama-3.1-70B-FP8, 250req) 129
6.65 Throughput vs. KV-cache utilisation (Tensor Parallel = 2, Llama-3.1-70B-FP8, 250req) 129
6.66 AI Inference Stakeholders Organized by System Layer 131

A.1 Sustainable Development Goals (SDG) 145

List of Tables

4.1 High-level project schedule. 52
4.2 Representative fixed cost components for infrastructure providers. . 56
4.3 Representative variable cost components for infrastructure providers. 57
4.4 Illustrative operating points at saturation under this study's benchmark conditions. 60

5.1 Overview of selected models used for benchmarking 63
5.2 Benchmark setup and hyper-parameters for the concurrency sweep . 66
5.3 Concurrency-sweep configurations for each of the selected models . 67
5.4 Most relevant Prometheus metrics to record for the concurrency benchmarks 68
5.5 Sequence-length benchmark matrix 70
5.6 Benchmark setup and hyper-parameters for the sequence-length experiments 71
5.7 Most relevant Prometheus metrics for the sequence-length benchmarks 72
5.8 Benchmark setup and hyper-parameters for the intra-node (tensor-parallel) scaling study 74
5.9 Run-specific configuration for the TP = 1 experiments 74
5.10 Run-specific configuration for the TP = 2 experiments 75
5.11 Key Prometheus metrics used for the intra-node scaling benchmarks (tensor parallelism 1 vs 2). 76

Listings

5.1 Example of OpenAI client pointed to the local vLLM endpoint and used to issue a request. 82
5.2 Parsing benchmark configuration parameters inside the workload generation subsystem. 86
5.3 Example of the orchestrator invoking the workload generation subsystem with parameters defined as variables. 87
5.4 Example slice of the `metrics.csv` generated for one benchmark run; actual files include hundreds of metrics and span the entire run duration. 90
5.5 Parsing Prometheus metrics and dynamically constructing a union header to capture intermittent metrics. 92

Chapter 1

Introduction

Over the past few years, large language models have transitioned from research artifacts to everyday tools that people consult for work, learning, and entertainment. Adoption has been rapid and sustained, and platforms built on generative models now serve very large user populations at the same time, a trend that continues to accelerate as new features and integrations expand the addressable audience [1]. As usage has grown, expectations have risen in parallel, because users who experience fluid interaction in one application come to expect similarly low waiting times and high availability in every application. The practical meaning of this shift is simple. A service that aspires to be widely useful must remain responsive under heavy demand and must deliver consistent quality regardless of time of day, geographic mix, or traffic spikes.

Meeting that standard depends on the capacity and efficiency of the underlying infrastructure. Modern generative systems are served most effectively on fleets of recent Graphics Processing Units that can process the linear algebra that is at the heart of inference, in a parallel manner. These GPUs reside in data centers, which means that the path from a user's prompt to a model's response runs through these facilities. Because these facilities must sustain both peak throughput and steady low latency, operators need to provision clusters that combine compute, memory, networking, and storage at significant scale. The consequence, however, is a cost structure that is dominated by hardware acquisition and by the recurring expenses of operating that hardware at high utilization. For example, public disclosures and reporting show that the largest providers have raised capital expenditures substantially to expand capacity for AI workloads, while unit prices for cutting edge accelerators remain in the tens of thousands of dollars, which magnifies the financial impact of even modest inefficiencies [2, 3, 4].

Because cost scales with capacity and demand is volatile, the economic viability of serving hinges on how efficiently each unit of hardware is converted into useful work. If resources are not used effectively, then the operator must deploy more GPUs to deliver the same user experience, and that choice in turn raises capital needs and depresses margins. If the operator chooses not to expand capacity, then the service absorbs the inefficiency as longer queues and slower responses during busy periods, which manifests as higher time to first token and elevated tail latencies that users perceive as unresponsive behavior. Either path has a direct business consequence. Higher hardware requirements increase cash outlays and extend payback periods, and degraded responsiveness leads to lower engagement and lower conversion for downstream products that depend on timely answers. At scale, both paths can coexist, which is why companies can report strong revenue growth while still facing sustained pressure on profitability from the infrastructure required to serve these workloads globally [5, 2, 3].

This tension between demand, cost, and user experience becomes sharper as adoption widens. More users translates into more concurrent sessions, and more concurrent sessions implies a larger aggregate working set moving through the serving stack at any point in time. Since users now expect conversational interactions rather than single shot prompts, sessions often persist across multiple turns and often include long contexts, which further increases the resources required to keep the experience responsive. The result is a system that must deliver both high throughput and low latency under workloads that are bursty and heterogeneous.

In this environment, the key question is how to maintain fast responses while keeping the cost envelope sustainable as usage scales. This is a widespread challenge in the industry, visible in the financial trajectories of the sector. For example, hyperscalers have reported sustained increases in capital expenditures to support AI infrastructure, which reflects both the need to build out new data centers and the need to upgrade existing facilities to higher power and cooling densities [2, 3]. At the same time, leading providers report strong revenue momentum while acknowledging the intensity of investment required to train and to serve modern models [5]. These signals point to the same underlying dynamic. Demand for AI services is large and growing, and the bottlenecks that limit efficient serving have system wide economic effects. If left unaddressed, those effects propagate outward from the data center into the product, where they appear as slower replies, stricter usage policies, and higher prices that can narrow the set of users who benefit from the technology.

The problem statement that follows from this chain of reasoning is clear. As generative systems continue to scale to larger audiences, operators must deliver interactive performance to millions of concurrent users in a way that keeps per

request cost within a sustainable range. The question is not whether further hardware investment is possible, but whether the relationship between demand and infrastructure costs can be managed so that growth strengthens the service rather than undermining its economics. Solving this problem is essential for AI-powered systems to remain viable as more people rely on them every day.

Chapter 2

Description of Technologies

This chapter presents the various technologies used in the development of the project. The entire system rests on a stack that ranges from the processing hardware, essential for running large-scale AI models, to the automation tools that ensure the reproducibility of the experiments.

Specifically, the solution is organised into three major functional blocks:

- **Execution and infrastructure**: GPU hardware and the interconnect network that determine computing capacity and scalability.
- **Processing**: The inference engine and model frameworks that allow large language models to be loaded, optimised and served.
- **Management and observability**: Workload generation, telemetry, automation and data analysis, all essential for measuring performance and drawing conclusions.

The technologies that make up each block are described in detail below.

2.1 NVIDIA H100 NVL

The NVIDIA H100 NVL is a dual-GPU inference accelerator built on the Hopper™ architecture and optimized for large-language-model deployment.

Each PCIe Gen5 card houses two H100 GPUs, each equipped with 94 GB of high-bandwidth HBM3 memory, for a combined total of 188 GB of on-board memory.

For this project, one NVIDIA H100 NVL's was utilised.

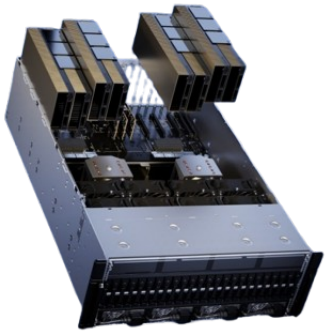


Figure 2.1: NVIDIA H100 NVL

Figure 2.2: NVIDIA Logo

2.2 vLLM Inference Engine

vLLM stands for vectorized Large Language Model. It is a high-performance and memory-efficient inference engine built to accelerate the deployment of large language models.

An inference engine is a runtime system responsible for executing trained models and generating outputs in response to user inputs. It manages the scheduling, memory usage, and compute resources needed to serve inference requests efficiently.

Designed with throughput and scalability in mind, vLLM addresses key performance bottlenecks in LLM serving by enabling faster, more efficient inference. Originally developed as part of the open-source BentoML ecosystem, it is engineered to make large-scale model serving practical and cost-effective.

Figure 2.3: vLLM Logo

2.3 Hugging Face

Hugging Face is an open-source platform and research community dedicated to natural-language processing and generative artificial intelligence. Established in 2016, it maintains the Hugging Face Model Hub, a public repository where thousands of pretrained checkpoints, tokenizers and configuration files are versioned under permissive licences.

HUGGING FACE

Figure 2.4: Hugging Face Logo

The Transformers library maintained by Hugging Face provides a uniform Python API that downloads, validates and initialises models with a single function call. When combined with vLLM, this interface allows checkpoints to be streamed directly into GPU memory, avoiding intermediate conversion steps and eliminating precision loss.

For this project, every open-source large language model required for the benchmarks was retrieved from the Hugging Face Model Hub and subsequently deployed on the H100 cluster to serve inference.

2.4 Prometheus

Prometheus is an open-source monitoring and alerting toolkit that was originally developed at SoundCloud in 2012. It collects metrics by periodically scraping HTTP endpoints that expose data in a text-based exposition format, and it stores the samples as time-stamped series that can later be queried or visualised.

Figure 2.5: Promethe Logo

In this project, Prometheus was leveraged through vLLM's built-in integration with the prometheus-client library, which automatically starts an HTTP endpoint at localhost:8000/metrics by default. This endpoint exposes a real-time list of performance metrics that include token throughput, request counts, inference latency, and KV cache usage, all formatted according to the Prometheus exposition standard. During benchmarking, this endpoint was scraped at regular intervals to collect and log these metrics for later analysis. Using Prometheus in this way provided significant benefits for the project, like real-time observability and detailed insights into performance bottlenecks without requiring a full monitoring infrastructure.

2.5 Python

Python is a high-level, interpreted, object-oriented programming language. It was created by Guido van Rossum and first released in 1991. Known for its clear syntax and clean design, Python has become one of the most widely used programming languages due to how easy it is to read, learn, and use.

In addition, Python stands out for its flexibility and can be applied in a wide variety of domains — from web development and data analysis to machine learning and automation. Its strong and active community provides extensive libraries and frameworks that speed up development and enable access to state-of-the-art tools

and techniques.

Figure 2.6: Python Logo

Python was selected for this project due to its clear and concise syntax, which allows developers to express complex ideas with minimal code. Its design philosophy prioritizes code clarity, reducing the likelihood of errors and making the implementation easier to maintain and extend over time.

2.5.1 Pandas

Pandas is a Python library designed for handling and analyzing structured data. It provides powerful and efficient data structures, such as the DataFrame, which allow users to organize, clean, and process data in a straightforward way. Pandas is widely used in data analysis tasks, making operations like filtering, grouping, and transforming data both simple and efficient.

In this project, Pandas has been essential for accessing and manipulating information related to hardware performance metrics collected during the benchmarking of LLM inference. Its capabilities have enabled the transformation of raw data, the filtering and selection of specific subsets, and the handling of missing or anomalous values, among other data processing tasks.

Figure 2.7: Pandas Logo

2.5.2 Matplotlib

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It provides a flexible and expressive API that allows users to generate a wide variety of plots, such as line charts, bar graphs, scatter plots, histograms, and heatmaps. Its pyplot module offers a simple interface for quickly building visualizations, making it a popular choice for data exploration and reporting.

In this project, Matplotlib has been used to visualize performance metrics, allowing for clearer interpretation of trends, comparisons between models, and identification of anomalies during LLM inference benchmarks.

Figure 2.8: Matplotlib Logo

Chapter 3

State of the Art

This chapter aims to give an overview of the research landscape most significant for understanding present-day LLMs. The discussion spans both the mathematical principles that govern LLM behavior and the practical realities of deploying these systems at scale. Once this foundation is laid, recognizing the challenges that surface in production will become clearer, leading to a better understanding of how the conducted benchmarks mirror those challenges.

Drawing on the broad corpus of LLM research, this chapter concentrates on the aspects most relevant to the objectives of the study. The chapter first opens with the life cycle of an LLM, tracing the path from model training to real-time inference. It then offers an intuitive look at the transformer architecture that underpins nearly every state-of-the-art model and explains its role in the recent surge of AI capability. Next, it examines the physical infrastructure that supports these models, including data centers, accelerators and networking, and follows with the principal strategies used to scale them efficiently. Finally, the chapter closes with a review of the key bottlenecks that emerge from the inner workings of LLMs, preparing the ground for the performance evaluation that follows.

3.1 Stages in the Life Cycle of an LLM

The life cycle of a large language model can be differentiated into two principal phases: training and inference [6]. Each phase presents distinct requirements that lead to different practical implications, such as resource footprint, hardware layout, and optimization strategies. For example, training prioritizes aggregate throughput and massive parallelism, while inference focuses on low latency and responsiveness under fluctuating demand. This section will cover the main phases

of the LLM life cycle, revealing what they consist of and how each introduces distinct operational characteristics and system-level requirements.

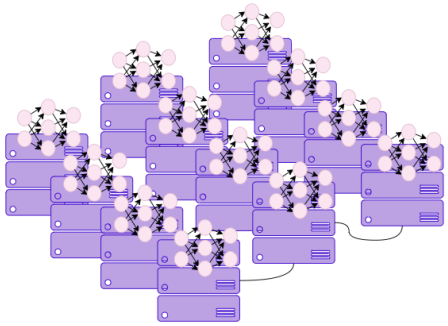


Figure 3.1: Large Scale Distributed Training

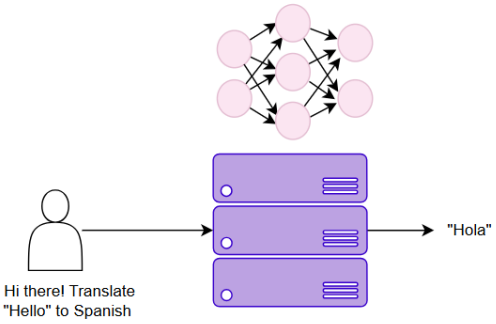


Figure 3.2: Inference

3.1.1 Distributed Training

The objective of training a large language model is teaching it to predict what comes next in a piece of text by learning from a vast dataset, typically ranging hundreds of billions to a few trillion pieces of text, where each piece could be a word, part of a word, or even a single character [7]. In a simplified view, the model is a system comprising a vast collection of numbers, called parameters, organized within a specific architecture (including attention mechanisms, feed-forward networks, and layer normalization) [8]. An intuitive way to think about these parameters is as turning knobs that the model can tune to adjust its own behavior, in a manner that improves its outputs. This is similar to how a musician might tune their instrument to produce the right notes. These parameters serve various purposes, such as representing token embeddings or weights in different layers of a neural network [9][10].

At the start of the training process, each parameter is initialized to a specific value, following schemes designed to prevent training problems. Initially, the model produces essentially meaningless outputs as it has not had the opportunity yet to learn any patterns in human language.

To learn to generate appropriate outputs, the model undergoes an iterative process of refinement. During each iteration, the model processes the input text and attempts to predict what comes next by calculating values through its neural network layers. The model generates a prediction in the form of probability scores for every possible word or character it knows. This prediction is then compared against the correct answer using a mathematical function, known as the

cross-entropy loss, which measures the accuracy of the prediction[11]. Using this measurement of accuracy, the model automatically adjusts its internal parameters through a mathematical optimization technique called backpropagation, gradually improving its ability to make accurate predictions in future iterations [12].

In order for a model to learn what qualifies as the correct output, several training paradigms may be employed. In practical systems, these paradigms are often combined sequentially, rather than applied in isolation, to take advantage of their complementary strengths.

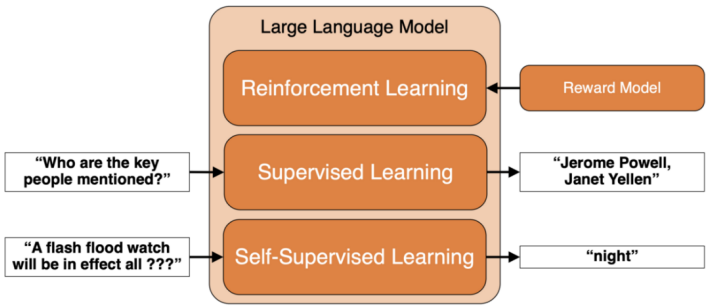


Figure 3.3: Learning stages in model training

A common initial stage in the training process of large language models is self-supervised learning [13]. This approach allows the model to learn from raw, unlabeled data by generating supervision signals directly from the data itself. Specifically, a portion of the data is treated as the input, while another portion is designated as the target output. The model is trained to predict the target based on the input.

In the context of natural language, self-supervised learning typically takes one of two forms. Masked language modeling, used in models like BERT, involves hiding certain tokens within a sentence and asking the model to predict them [14]. Next-token prediction, employed by models such as GPT, involves providing a sequence of tokens and training the model to predict the following token [15]. Both methods enable the model to process incomplete text and optimize its predictions by comparing them to the actual tokens in the dataset, improving results with each iteration.

By training on that raw text, the model builds a broad internal understanding of grammar, facts, and reasoning patterns. The key advantage of self-supervised learning is that these representations are learned without the need for human-annotated labels, which are significantly expensive. Instead, raw text that is plentiful and cheap can be used instead. Therefore, it makes this an ideal starting

point for model training, because it allows the model to develop rich internal representations using large volumes of inexpensive text, making it feasible to train the model on massive datasets collected from the internet, without the need for costly human annotation.

To further improve the model's performance on specific tasks, a subsequent training phase known as supervised learning is often employed [16]. In this phase, each training example consists of an input paired with a corresponding output label, typically provided by human annotators. However, creating high-quality labeled datasets often requires the participation of skilled annotators and, in some cases, multiple rounds of review to ensure accuracy and consistency, which is more costly. That is the reason why it is usually only applied after self-supervised learning has created a good model foundation for language understanding. Nonetheless, this second phase is needed in order to improve the ability of the model to complete well-defined tasks where there's a clear, correct answer.

Even with this added supervision, though, the model does not always behave in ways that align with what people actually want. This becomes especially important in open-ended or ambiguous situations, where many answers might be possible, but only some are truly helpful, polite, or even safe. For that reason, a third training stage known as reinforcement learning is often introduced [17]. Unlike supervised learning, reinforcement learning does not rely on explicit labels. Instead, the model interacts with an environment and learns from feedback in the form of rewards or penalties. Rather than being told exactly what to say, the model explores different outputs and gradually learns which types of responses are preferred, based on whether they lead to higher or lower reward signals.

To summarize, training a language model involves three main stages. Self-supervised learning builds a broad understanding of language using raw text. Supervised training then teaches the model to follow specific instructions using labeled examples. Finally, reinforcement learning helps the model align its responses with what people actually prefer, especially in more open-ended situations.

While understanding how the model learns is key, it is also important to look at how this training is actually carried out in practice, given the massive scale of today's language models. Training is performed offline and typically only a few times, until the model reaches an acceptable level of performance. Because the model must process vast amounts of text to learn meaningful patterns, training times can take from a few weeks up to multiple months [18]. To reduce this time, engineers focus on maximizing throughput, which refers to how much text the model can process per second during training.

The training of these LLMs is done offline, and it happens in a punctual manner,

a few times, until an acceptable performance is achieved. The huge amounts of data that the model needs to ingest in order to obtain insightful patterns can lead to very big training times, so engineers try to optimize for throughput so that the training is done in less time. This is done by using thousands of computing nodes in parallel, which work together in a distributed manner as if they were a single machine [19]. To function cohesively, these nodes must be interconnected through a high-speed network, making bandwidth one of the most critical requirements.

3.1.2 Production-Grade Inference

Once the model has been successfully trained, the next phase in its lifecycle is called inference. Inference refers to the stage where the trained model is used to generate outputs based on new input data that was not seen during training. For example, using the patterns it has learned, the model may be asked to translate a sentence or to explain a particular concept.

The nature of the workload during inference is fundamentally different from training. Unlike in training, the model does not update its internal weights. Instead, it uses the same fixed parameters to produce outputs consistently for each input. In many language models, the response is generated step by step, where each new part of the output depends on everything that has been produced so far. This approach is known as autoregressive, and it works much like writing a sentence one word at a time, where each word is chosen based on the ones that came before. By building the response step by step in this way, the model can stay consistent with the context and gradually shape a coherent and meaningful output [20].

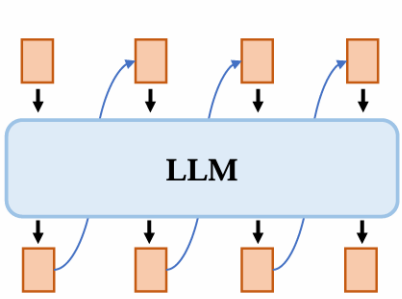


Figure 3.4: Autorregressive mechanism in LLM inference

Additionally, the way training and production inference are executed on hardware differs significantly. As previously mentioned, training focuses on processing vast amounts of data in parallel across thousands of machines, with the goal of maximizing throughput and completing the training process in the shortest possible time. Inference, however, introduces a different set of constraints. Because of the

autoregressive nature of many language models, each part of the output must be generated sequentially, with every new element depending on what has already been produced. This limits the benefits of parallelism. According to Amdahl's Law, when most of a process is inherently sequential, adding more computing resources results in diminishing returns. Distributing a single inference task across many devices increases communication overhead, which can outweigh any speed-up gained from additional hardware [21].

As a result, it is generally more efficient to serve many users by running multiple independent model replicas on individual GPUs, rather than relying on one big, tightly coordinated multi-GPU setup.

It is of great importance to take into account how, in real-life deployments, the role of inference is to serve the requests of real users interacting with the model in real time. These systems have recently grown to massive scales, often handling thousands of requests every second [22]. In this context, inference systems must also be carefully designed to optimize for latency and fault tolerance, since all of these factors directly impact how fast and reliable the model feels to the end user.

In addition to throughput, inference systems must also optimize for other critical factors such as latency and fault tolerance, all of which directly affect the quality and reliability of the deployed model.

These factors become especially important when considering that, unlike training, inference runs continuously. Every time someone asks the model a question, requests a translation, or uses it in an application, inference is taking place. This turns inference into a recurring cost that depends on how often the model is used, how many users it serves, and how much computing power is needed to generate each response. While training represents a significant one-time investment, inference is where the ongoing costs accumulate, so as usage scales, these recurring costs can grow substantially and, in the long run, may even surpass the cost of training itself.

3.2 Transformer Architecture

The previous chapter explored how large language models are trained and served, highlighting the distinct computational demands of training and inference. To understand why these demands grow so quickly, it helps to look inside the models themselves. Most of today's leading language systems are built on the Transformer architecture, whose design choices largely determine both their power and their scaling challenges.

3.2.1 Why transformers have led the AI boom

During the revival of neural-network research in the mid-1980s, much attention was focused on the multi-layer perceptron (MLP). This feed-forward network accepted a single, fixed-length input vector and delivered promising results on small image tasks, such as classifying 16 × 16-pixel characters. MLPs, however, struggled with sequential data such as speech, language, and sensor readings. Because an MLP processed all input positions at once and had no internal memory, it could not link the current step to what came before, so it failed to capture the patterns that sequential data needed [23].

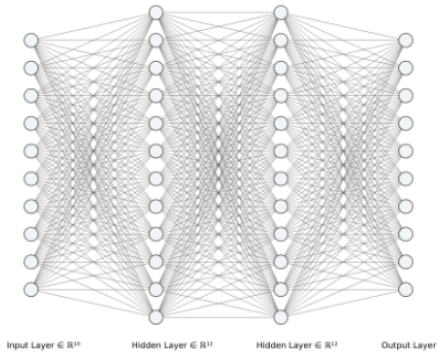


Figure 3.5: Multi-layer Perceptron

Researchers soon realized that solving sequential tasks would need a new kind of network, one that could remember what happened a moment ago in order to figure out what comes next. The solution became the recurrent neural network (RNN). From the early 1990s through the mid-2010s, RNNs were the standard tool for tasks that involved text or audio [23]. They powered language modeling for text generation, speech recognition, sentiment analysis, image captioning, and even early question-answering systems, all by using their built-in memory to link each step of a sequence to the one before it. What made this model so widespread for these tasks is the capability of processing causally ordered data, and being able to accomodate for different lengths of inputs without having to change the underlying design. An RNN reads information much like a person reads a sentence, one element at a time. At each step it acceptes the current symbol (for example, a character, a word, or a short slice of audio), combines it with what it has already stored in memory, and produces an intermediate output called the hidden state. This hidden state then moves forward to the next step, allowing the network to stitch context across the entire sequence.

Although this mechanism captures short-range patterns effectively, training plain

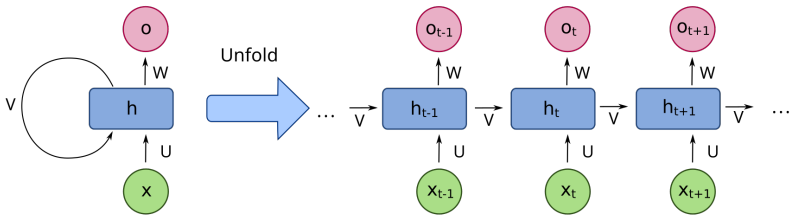


Figure 3.6: Recurrent Neural Network

RNNs revealed a deeper limitation. One of the main challenges was that the hidden state had a fixed dimensionality, meaning the network had to compress all relevant information from the past into a space of constant size. Trying to store an entire sentence in a single fixed-width vector is like trying to fit a novel onto a sticky note;important details are inevitably lost as the sequence grows longer.

When the relevant history extends beyond what the hidden state can meaningfully retain, the network struggles to preserve useful context. This leads to issues during training, where gradients either vanish or explode, making it difficult to learn long-range dependencies [24]. For example, in a summarization task involving long input text, an RNN might fail to maintain the necessary context across time steps, resulting in incomplete or low-quality summaries.

These limitations motivated the development of more advanced RNN architectures, such as the Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU), which aimed to better preserve long-term information [25]. While these variants offered improvements, they still fell short of delivering a breakthrough in handling truly long-range dependencies.

In 2014, Bahdanau, Cho, and Bengio introduced a new idea called attention [26]. At the time, the common approach to sequence tasks like translation was to use RNNs that first processed the entire input sentence and then compressed it into a single vector, which the model would use to generate the output. But this method had a major limitation: it assumed that every part of the input was equally relevant for predicting each word in the output. In reality, different output words often depend more on certain parts of the input than others [27]. The researchers proposed a better approach. Instead of using the same compressed vector for every step of the output, the model could create a new one each time, focusing only on the parts of the input that mattered most in that moment. This idea, called attention, allowed the model to weigh each piece of the input differently depending on what it was trying to predict.

The details of how attention works will be explored in the next sections, but for now, it is enough to understand this: attention solved the problem of forcing the model to squeeze all input information into a single vector. Instead, it gave the model the ability to look back and selectively focus on what truly mattered, step by step.

Even with all these improvements, RNNs still faced a major limitation that made them difficult to use at large scale. Because they processed input one step at a time, each step had to wait for the one before it to finish. This meant that the model could not take full advantage of modern hardware like graphic processing units (GPUs), which are designed to process many things in parallel. While some parallelism was possible across different sequences, the model still had to move step by step through each one, which made training slower.

This became a problem when trying to scale up. In real-world systems that involve huge amounts of data or serve large numbers of users, efficiency matters. RNNs struggled to keep up because their design made it hard to speed things up using the kinds of parallel computing that modern infrastructure relies on. This bottleneck helped motivate the search for new architectures that could overcome these limits.

Everything changed in 2017 when a group of researchers introduced the Transformer architecture in the paper "Attention Is All You Need" [28]. At the time, most models for processing text were based on RNNs, which read one word at a time, in order. The Transformer took a different approach. Instead of reading words step by step, it looked at the entire sequence all at once, allowing the model to process the full input in parallel.

This was a major breakthrough, because it meant that modern hardware like GPUs, which are built to handle many computations at the same time, could be used much more efficiently [28]. Training became faster, which made it possible to build larger models without waiting weeks for each experiment to finish.

The Transformer still relied on the idea of attention, which had been introduced a few years earlier, but it placed it at the center of the architecture. The key innovation was finding a way to make attention work without relying on recurrence, which made parallel processing possible. In doing so, the Transformer combined the best of both worlds: it kept the ability to focus on the most relevant parts of the input, while being far more scalable than earlier approaches like RNNs.

It was the development of the Transformer architecture that opened the door to the current AI boom. By allowing models to process entire sequences of words in parallel rather than one at a time, it became possible to train much larger models

in much less time, while taking full advantage of modern hardware like GPUs. This shift made it feasible to scale model size dramatically without training becoming unbearably slow or expensive. Once those technical roadblocks around training speed and context handling were relaxed, scale took over.

In 2020, OpenAI released GPT-3, a 175-billion-parameter Transformer model that could perform a wide variety of language tasks just by seeing a few examples written in plain text [29]. This ability, called in-context learning, showed that one single model could adapt on the fly to translation, summarization, question answering, and more, all without task-specific training.

Two years later, ChatGPT reached one hundred million users in just eight weeks, becoming the fastest-growing consumer application in history [30]. This moment kicked off a wave of generative AI tools, research, and investment across the world. By combining flexible attention with efficient parallelism, the Transformer delivered the two key ingredients needed for progress: the ability to understand long-range context and the practicality to train at massive scale. Together, those advances turned what had once been a research idea into the foundation of today's AI revolution.

3.2.2 Architectural breakdown of Transformers

Before diving into how Transformers generate responses, it helps to take a step back and look at how they are built. This section walks through the main parts that make up a Transformer model, starting with how text gets prepared and ending with how the model produces its output. Each part has a specific role in helping the model understand context, recognize patterns, and generate language that makes sense.

When a prompt is typed into a Transformer-based model, the first step is to break this non-fixed length text into discrete symbols that the model can reason about [31]. Splitting text into individual characters is too fine-grained, as each character carries limited semantic meaning on its own. In contrast, using full words can be too coarse, since new, rare, or misspelled words appear frequently, resulting in an impractically large vocabulary for the model to handle. Algorithms like byte-pair encoding (BPE) or WordPiece found a solution by splitting phrases by frequently recurring groups of letters, known as tokens [32]. This way, common words like "network" end up as single tokens, whereas rarer or morphologically complex items like "generalisation" are split into pieces such as "general", "isa", "tion". On average, one token tends to be about four characters long. For simplicity, this text might sometimes refer to tokens and words interchangeably, although they are not exactly the same thing.

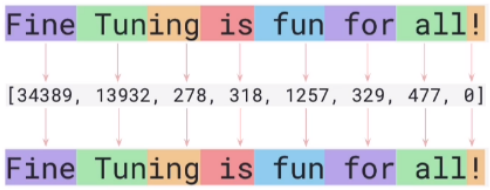


Figure 3.7: Tokenization

This tokenization technique yielded a vocabulary that is compact enough for the neural network's parameters to handle, yet expressive enough to reconstruct any string unambiguously by concatenation. So every time a user inputs a prompt, this one gets split into tokens, and mapped to the token ID that represents each one. For example, the sentence "Transformers rock!" might become the sequence 3201, 1059, 14, 2, where the final value 2 represents an end-of-sequence marker, also represented as an <eos> symbol.

Splitting the prompt into a sequence of token IDs is not enough, though. Given that each token has been assigned an arbitrary numerical ID, which carries no semantic meaning on its own, the meaning of the word is lost. In order to perform well, the model needs a representation of words that allow to understand relationships between other words. In a single dimension, every token is forced to live on a straight number line: one coordinate, no angles, just 'larger' or 'smaller'. Imagine trying to arrange the four tokens: woman, man, girl, and boy on that line. They differ along at least two meaningful axes, gender and age, but the line can honor at most one of them. In this case, if woman and man terms are placed closer together, and girl and boy are placed in the other side to represent age, the girl would be stranded away from woman even though she is also female, and same with the boy and man. No matter how you shuffle positions, it is not possible to encode two different types of relationships (age and gender) that the language demands in a single dimension. However, representing these tokens in a two-dimensional space would resolve the issue: By drawing the x-axis for gender and the y-axis for age, suddenly orthogonality and Euclidean distance work together to encode every relationship correctly.

In real life, words can share hundreds of thousands of subtle relationships with one another. To capture these patterns in a form that a model can reason about, each word is represented as a high-dimensional vector, commonly known as an embedding. The dimensionality of these vectors, denoted as d, is chosen based on what was considered sufficient for the model's capacity and task. By moving beyond a single dimension into dozens or even hundreds, the model gains more

degrees of freedom to encode meaning. In this multi-dimensional space, properties such as vector length, direction, and the angles between vectors all carry useful information. This allows the space to bend and organize itself so that similar words naturally cluster together based on how often they appear in similar contexts.

A quick thought experiment can show the value of having extra room in a high-dimensional space. Suppose that words such as king, queen, prince, princess, man, woman, boy, girl, lion, and lioness are placed into a space defined by four carefully chosen axes: gender, age, social rank, and species.

Suddenly, simple vector arithmetic can be used to capture relationships that humans recognize intuitively: the displacement king $\rightarrow$ queen is almost identical to man $\rightarrow$ woman, while king $\rightarrow$ prince parallels man $\rightarrow$ boy. Even cross-species analogies emerge: king $\rightarrow$ lion and queen $\rightarrow$ lioness line up because the "royal" dimension mirrors "apex animal." What humans achieve informally with language (such as grouping concepts along many overlapping criteria), the embedding formalizes with vectors, allowing subsequent layers of the Transformer to manipulate meaning with nothing more exotic than dot products and matrix multiplications.

The more dimensions, the more degrees of freedom exist in order to represent the complex semantical relationships between words. Two vectors can be nearly parallel to signal near-synonymy, almost orthogonal to signal irrelevance, or opposed to mark antonymy, all without forcing unrelated pairs to collide. In that sense, dimensionality is not the whole story, since training data and model depth also matter. Nonetheless, without a sufficiently roomy space the other ingredients would have nowhere to write the subtle patterns humans interpret as semantic knowledge.

This mapping of tokens to embeddings from the input prompt is performed in parallel, rather than one token at a time. As a result, the model receives a collection of embeddings (vectors of dimension d) corresponding to the initial prompt, which it must then process to generate an output. However, since all tokens have been processed simultaneously, the model has no inherent way of knowing the original order of the embeddings. They are effectively unordered.

This presents a problem, because it is clear that the order of words in a sentence is essential for understanding meaning. For example, the sentence "Dog bites man" conveys something very different from "Man bites dog." To address this, a positional encoding is added to each embedding before the sequence enters the first block of the Transformer architecture. These encodings provide information about the position of each token in the sequence.

Once the token embeddings have been combined with their respective positional signals, the prompt has been fully translated into a format the model can understand. From that point onward, the sequence can be passed through the model to generate the desired output.

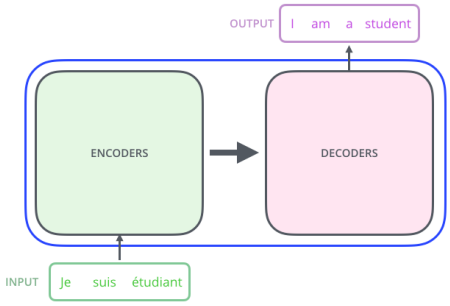


Figure 3.8: Encoder-decoder Transformer

Now here is an overview of what blocks are inside of the transformer architectural model. The 2017 Google version of the Transformer architecture, often referred as the Vanilla Transformer, encompased two major blocks, called the encoder and the decoder [28]. The encoder's role was to process the entire input sequence in parallel, in order to gain a context vector which is a representation of the contents of the input. On the other hand, the decoder's role was to generate the output, one token at a time. However, in recent years the decoder-only transformer was introduced, like for example gpt-2, and it proved to perform to the same standard as the vanilla transformer in text generation tasks, after adjustments and fine-tuning, so it became the de-facto architecture for LLMs from there forward [33]. Most state of the art models, like Deepseek, OpenAI's o3, LLamas, Grok etc. all have the decoder-only architecture, so this is the one that will be explained further [34].

As previously explained, the input of the transformer model is a prompt, turned into embeddings, which are all taken at once, in a parallel manner. Because of this, positional encodings are added to establish the correct order of words of the sequence [35]. While these embeddings capture some semantic relationships between words, they lack context sensitivity. This means the same word will have the same embedding regardless of its context within a sentence or document.

The prompt is then forwarded through all of the blocks of the transformer model, and the output, is a probability distribution. For GPT models, the output is the probability of each token being the next token in the sequence. For example, if the vocabulary of a model were to be composed of around 50,000 words, the model

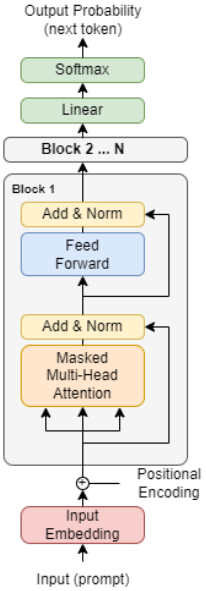


Figure 3.9: Decoder-only Transformer

would output at each new step the probability that each of them is the next token. For example, if the input is " I am going to the " the next token is more likely to be the word "park" than "ate" or any other words of the vocabulary.

In modern large language models like GPT, the prompt is extended one word at a time. The initial prompt is inserted, and the first word of the answer is generated. This word is then appended to the original prompt, forming a slightly longer input. The new prompt is passed through the model again to predict the next word. In this way, the full full output will be constructed one token at a time through an iterative process known as autoregression.

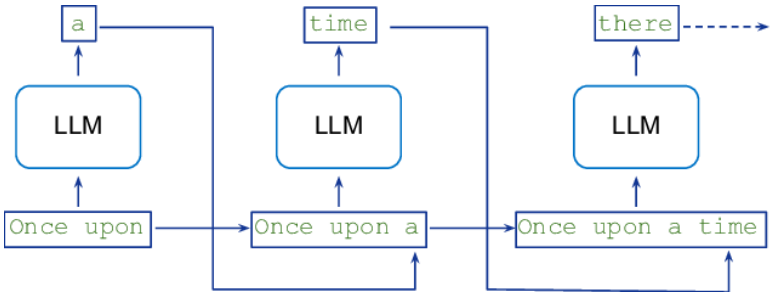


Figure 3.10: Autoregressive decoding

As a note, during inference, there are several techniques for selecting the next token based on the model's output. Since the model produces a probability distribution

over all possible next tokens, one option is to sample from that distribution. This introduces an element of randomness, so that even if the same prompt is given multiple times, the generated output may differ each time. Another option is to always select the token with the highest probability, which makes the model's behavior fully deterministic.

Inside each of the blocks, lies two fundamental components. The first one is the attention block, and the second is the feed-forward network [36].

The attention block is what makes the transformer powerful. It is the block that enables the model to understand the context and nuances in languages, making it better at reasoning.The core idea is inspired by how humans process language. When reading a sentence, meaning is not assigned to each word in isolation, but rather in relation to the surrounding words. For instance, the word "second" can take on different meanings depending on context. In the sentence "The light turned green within a second," it refers to time, while in "She finished in second place in the race", it refers to position. The role of the attention block is to take each original embedding, which initially contains no context, and transform it into a richer, more informative version that reflects the meaning of the word within its specific context. Following the "second" example, after passing through the attention block, the embedding vector for the word "second" in the first sentence would shift to a region in the embedding space closer to vectors like "time" or "clock." In contrast, in the second sentence, that same embedding would move toward vectors such as "ranking" or "position." This demonstrates how proximity in embedding space can reflect semantic similarity. These contextually enriched embeddings provide the model with a better understanding of the relationships between words, allowing it to reason more effectively by capturing the subtleties and nuances of language [37]. However, this attention block also introduces practical complexities that are highly relevant to this thesis and will be discussed in detail in later sections.

Finally, the attention mechanism enriches the embeddings with contextual information, these vectors are passed through a feed-forward layer, which is a simple neural network applied in parallel to each embedding. The role of this layer is to process the context-aware embeddings and perform deeper reasoning on them. It can be thought of as the part of the model that interprets and stores knowledge. Through training, this component can learn factual information such as the capital of Spain being Madrid, the start date of World War II, or how to write code in Python. In addition to the attention mechanism and feed-forward layer, another component called layer normalization also occurs before and after these main sub-layers. Its role is to stabilize training by normalizing the input values, which helps the model converge more reliably. While important for performance,

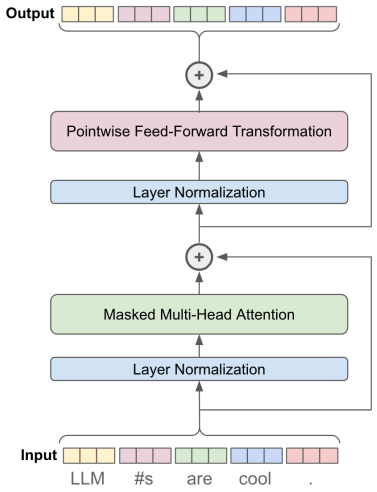


Figure 3.11: A decoder-only Transformer block

it will not be the main focus of this section.

These two main sequential steps, the attention mechanism and the feed-forward layer, are grouped into a module known as a Transformer block [36]. This block is repeated N times to increase the model's capacity to form complex and nuanced abstractions, which ultimately improves its performance. Once the input has passed through all N blocks, the model produces the next token in the sequence. The updated prompt, now including the newly generated token, is fed back into the model, and this process continues iteratively until the model signals the end of the response by producing a special end-of-sequence token (<eos>). However, in some cases, the output may stop before this token is reached if the language model provider enforces a limit on output length to manage computational costs. From the users' perspective, this may result in incomplete or abruptly ended responses.

3.2.3 The Attention Mechanism

The attention mechanism allows all tokens of the prompt to attend to all other tokens, in more or less measure based on how much relevance each other token poses. The goal is to take a word embedding, and through passing it through the attention block, to shift this vector embedding so that the new representation now takes account of the context around it. The question now becomes how the model determines which other tokens are most relevant to the current one. It does that by calculating, for each previous word, what its attention score is to the new one. The higher the value, the higher the relevance that token has with the new one.

The mathematical expression of attention is the following [28]:

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

Figure 3.12: Mathematical expression of the Attention mechanism

The mechanism has three main elements: the Query, the Key and the Value. These names come from the database terminology, because it has similarities with how words are searched in a database.

In the attention head, the model first receives the input token embedding, and generates three vectors called Q, K and V from it. These three elements are calculated by multiplying the input embedding through Wq, Wk and Wv [28]. These three matrices weights are generated during training the model, and remain unchanged once training is done. The Q vector can be thought to represent a Query, meaning a question of how this token relates to all other previous ones. This query is then compared against all keys of all previous tokens, and this generates attention scores for each of the previous tokens, that signify how relevant each one is to the given query. For example the phrase "My mother is pretty and she ..." the word "she" will calculate the attention score which each of the previous words, and in this case will yield a higher scores for the words "mother" and "my" because these are the most relevant to this new token. This comparison of Query and Keys is represented mathematically by a matrix multiplication, which in the formula is the dot product between Q and K [34].

Now that a list of attention scores for each key has been computed, the degree of relevance of the previous words to the current token can be understood. The next objective is to determine how much the original embedding should be adjusted in each direction to more closely resemble the keys that received higher attention scores. Much like in databases, where information is stored in key-value pairs, in Transformers the key is used to represent previous tokens in the sequence, while the value encodes how the original embedding should be altered to reflect similarity to that key. In the final step, each attention score assigned to a key is cross-referenced with its corresponding value. The attention score determines the proportion of influence, and the value provides the precise direction and magnitude of the adjustment. Together, these allow the original embedding to be transformed into a contextually enriched version that better reflects its relationship to surrounding tokens. By adding together all of the contributions from each of the keys and their attention score (a * V) to the original embedding, the result is a new embedding that has shifter closer to the previous tokens that were contextually relevant, gaining a richer expression of the word, therefore allowing the

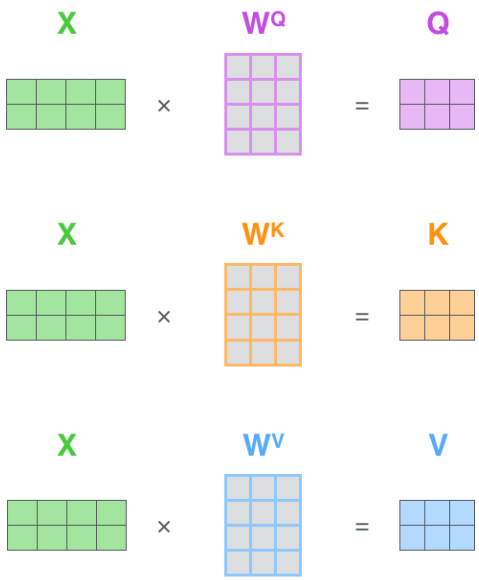


Figure 3.13: Query, Key and Value computation

model to reason better. Following the presented example, the word "she" in the sentence "My mother is pretty and she ..." by passing through the attention block, will shift in the high-dimensional space in a way that will make it a lot closer to the word "mother", a bit closer to the word "my" and a tiny bit closer to the word "pretty". How much it will move towards each of the other words will be determined by the resulting attention scores of each of them. A higher attention score will yield a higher movement towards similarity.

As a note, once the dot product between Q and K has been computed and the raw attention scores obtained, a softmax function is applied. This transformation is used to normalize the scores into a probability-like distribution between 0 and 1, allowing them to be interpreted as relative importance weights. Additionally, it has been observed that as the dimensionality of the K and V vectors, denoted as d_k, increases, the resulting values from the QK product tend to grow disproportionately large. To prevent these inflated values from overwhelming the softmax computation, the scores are scaled by dividing them by the square root of d_k. This scaling helps maintain numerical stability and ensures that the attention remains balanced across different positions [38].

In modern large language models, it is often assumed that a single token may hold multiple types of relationships with other tokens in the sequence. To capture these diverse relationships, multiple attention heads are used instead of relying on

$$head_i = Attention(XW_i^Q, XW_i^K, XW_i^v)$$

$$MultiHead(X) = [head_1, ..., head_h]W^o$$

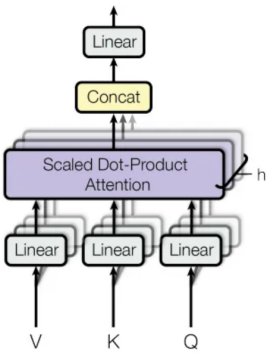


Figure 3.14: Multi-Head Attention

only one [39]. In this configuration, each attention head is assigned its own set of learned weight matrices, namely W_q, W_k, and W_v, which allows different aspects of the input to be processed in parallel. This setup enables distinct semantic patterns to be captured at the same time, leading to a richer and more nuanced understanding of the input.

3.3 From Theory to Execution: Infrastructure behind LLMs

Most major large language models are built upon the transformer architecture, which defines the mathematical operations guiding their behavior. However, understanding these operations in isolation provides only a partial view. It is equally important to examine how these computations are carried out by physical infrastructure, since that is where the practical challenges of deploying and scaling such models become apparent. The complexity of inference at scale does not arise solely from the model itself, but from the interaction between algorithmic design, hardware capabilities, and system-level coordination. To truly understand how LLMs work, it is necessary to connect the theory behind them with how they are actually run on machines.

3.3.1 Data Centers

When someone interacts with a large language model, for example by asking it a question or giving it a prompt, the response that comes back is not generated on their phone or laptop. Instead, the request travels across the internet to a remote facility known as a data center. That is where the real work happens.

These data centers are the physical homes of the powerful computers that make large language models possible. They achieve this by housing the machines and infrastructure needed to run heavy computations. While many types of data centers exist, some are designed specifically for artificial intelligence workloads. Additionally, data centers can be found in many parts of the world, and their location often depends on access to cheap electricity, reliable internet connectivity, political stability, and other factors. Given the scale of their operations, their environmental impact is becoming increasingly significant. Recent findings show that data centers accounted for more than 4% of total U.S. electricity consumption in 2024 [40].

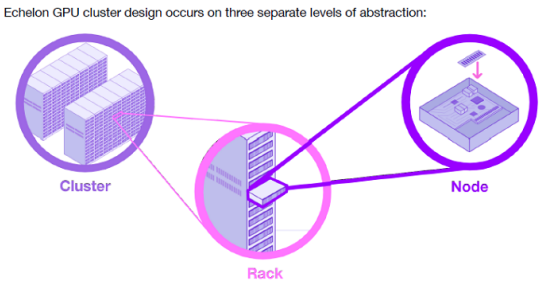


Figure 3.15: Abstraction Layers in a Data Center

Inside an AI data center, the infrastructure is carefully organized into distinct layers, each serving a specific function [41]. At the highest level of abstraction lies the cluster level, which involves planning the overall layout of the data center floor. This includes working within physical space constraints, managing power availability, and ensuring that the design will support the required computational capacity.

The next layer of design occurs at the rack level. A cluster is typically composed of multiple racks, and this stage involves determining the physical arrangement and exact placement of individual machines, or nodes, within each rack. It also includes planning how these nodes will be interconnected to function as a unified system.

Finally, at the node level, the previously defined computational and storage requirements guide the selection of hardware components. Each node typically

contains a central processing unit (CPU) and multiple graphics processing units (GPUs), which are interconnected through a high-speed fabric. The CPU is responsible for tasks such as booting and running the operating system, managing incoming requests or running background services. These types of tasks are often difficult to parallelize and involve many small, sequential decisions. Because of this, CPUs are particularly well suited for handling control-heavy workloads that do not benefit much from running on many processors at once.

However, in the transformer architecture that has been mathematically described in section 3.2.3, all computations can be mostly boiled down to a series of matrix multiplications, which are operations that are inherently parallel-friendly. To illustrate this parallelism, Consider two 2×2 matrices

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

Their product $C = AB$ has entries

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21}, \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22}, \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21}, \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22}. \end{aligned}$$

Each c_{ij} depends only on one row of A and one column of B, never on the values of any other element of C. Four independent compute units can therefore load their required row and column, perform the two multiply–add operations, and write their single result without waiting for the others. Because there is no data dependency among the outputs, each computation can be done in parallel. Therefore the computation scales cleanly across as many parallel lanes as the hardware provides. For this reason, GPUs are preferred over CPUs for running the model's forward pass. GPUs are specifically designed to handle large-scale parallel workloads, making them particularly well-suited for LLM inference. Within a single node, multiple GPUs can be used in coordination with the CPU, enabling significantly faster inference than would be possible with CPUs alone.

In data centers, a critical component of the infrastructure is the networking fabric, which manages communication between machines and nodes. A widely adopted topology is the leaf–spine architecture, a two-tier implementation of the classic Clos network design [42]. In this setup, leaf switches are installed at the rack level and connect directly to all nodes within their respective racks. These switches are typically implemented as Top of Rack (ToR) switches. To avoid a single point of failure, it is common to deploy multiple ToR switches per rack. This redundancy increases fault tolerance and enhances the overall reliability of the system.

However, nodes in different racks often need to exchange data. For this reason, each ToR switch is connected to one or more spine switches, which form a high-speed aggregation layer that interconnects all ToR switches in the cluster. This design ensures that traffic between any two racks can traverse the fabric by passing through at most one ToR and one spine switch, ensuring consistent latency and bandwidth across the entire cluster.

This design is highly scalable, as additional spine switches can be introduced to increase total network bandwidth without requiring a complete redesign of the fabric. It also offers path diversity and redundancy: if one spine switch fails, traffic can be rerouted through alternative paths, preserving connectivity and performance.

Both the nodes connected to the ToR switches and the ToR switches connected to the spine layer typically use Ethernet with RoCEv2 (RDMA over Converged Ethernet) or InfiniBand, depending on the cluster architecture [43][44]. In clusters optimized for peak AI training and inference performance, InfiniBand is often preferred due to its ultra-low latency and support for in-network compute. Meanwhile, RoCEv2 running over high-speed Ethernet links (400 or 800 Gbps) is increasingly popular in large-scale cloud deployments, as it builds on existing Ethernet infrastructure while still offering RDMA capabilities.

3.3.2 GPU Memory and Hardware Constraints

Understanding the structure of modern data centers reveals the physical limits that shape the deployment of large-scale AI workloads, such as constraints imposed by power, cooling, and network topology. However, within each rack and server, another layer of constraint plays a more direct role in model execution: the capabilities and limitations of the GPUs themselves. Even with high-speed interconnects and densely packed accelerators, inference performance ultimately depends on what each GPU can compute and, critically, how much data it can store in memory. The following section shifts the focus from the data center-level infrastructure to the local hardware level, examining current GPU memory capacities, how they interact with model weights and runtime data, and why they often become the primary bottleneck in serving large models at scale.

First, it helps to have a clear understanding of the high-end GPU landscape as of for 2025. NVIDIA is considered a pioneer in GPU technology, and has multiple products that vary in cost and performance, and categorized in three main architectures: the Ampere, Hopper and Blackwell architectures.

In 2020, NVIDIA released the A100, based on the Ampere architecture, which quickly became the go-to accelerator for both training and inference of LLMs.

In late 2022, the H100 was launched, based on the new Hopper architecture. It kept the same memory capacity but significantly increased both throughput and memory bandwidth. Then, in 2024, the H200 was announced as an upgrade to the H100, and in 2025 the Blackwell architecture has been launched as a predecesor of the Hopper, with their new B300, having x11 more throughput than H100 during inference [45][46].

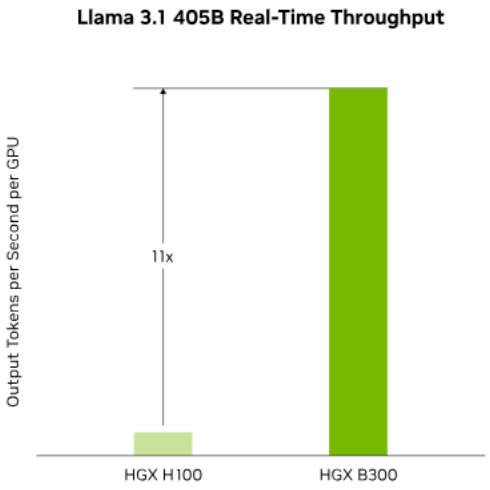


Figure 3.16: Performance comparison of LLM Inference with H100 and B300 done by NVIDIA

In order to understand what difference they have between each other, we need to introduce some terms that are relevant to performance in LLM inference at the hardware level. Cuda cores are the primary processing units of the GPU. Higher CUDA core counts generally translate to better parallel processing performance. Additionally, tensor cores are specifically useful for deep learning tasks, given that they are optimized for dense matrix multiplications. Therefore, a larger number of tensor cores directly translates to immediate throughput gains. Another very relevant parameter is the VRAM GPU memory. This is the memory available to store the model itself and the prompts that users send to the model. The more memory there is, the higher number of concurrent users can be served, improving throughput. However, the available memory is irrelevant without sufficient memory bandwidth. No matter how fast a GPU is at doing computations, if it must wait for the model weights and prompts to be retrieved form the GPU memory, there will be idle time that will increase latency.

Knowing this,

These performance figures are not just theoretical. OpenAI disclosed that GPT-4 training consumed roughly twenty-five thousand A100 GPUs over about 90 to 100 days [47]. Even at conservative cloud pricing, that footprint represents hundreds of millions of dollars in hardware time alone. when it comes to inference, OpenAI used a cluster of 128 A100s to serve the live model to the public, keeping enough capacity on hand for traffic spikes. In private efforts, the investment grows even larger. In a recent interview, Jensen Huang described how xAI assembled a supercomputer made up of one hundred thousand H100 GPUs in just nineteen days. At a cost of thirty to forty thousand dollars per H100, the total expense for GPUs alone ranges between three and four billion dollars [48]. Such numbers underline a basic fact: achieving competitive AI performance requires massive initial investment in infrastructure.

3.3.3 Quantization

When memory becomes the bottleneck, reducing the size of the data stored becomes a natural response. One of the most effective ways to do this is by lowering the precision of the numbers used to represent a model's weights. At its core, a transformer is a network of parameters, each a single number. These values are usually stored in 16-bit floating point (FP16) or, increasingly, in even smaller formats like FP8. Since memory usage scales directly with the number of bits per parameter, switching from FP16 to FP8 can nearly halve the memory footprint [49]. This enables serving larger models or handling more simultaneous requests without increasing hardware.

Figure 3.17: Quantization example

This reduction in precision is called quantization. In practice, applying quantization trades some accuracy for gains in latency and throughput. Lower precision numbers are faster to move, faster to multiply, and take up less memory per token. The obvious concern is that shrinking precision might hurt output quality, however, larger networks seem to tolerate quantization better, maintaining quality while gaining speed.

3.4 Scaling for Real-World Inference

3.4.1 Inference SLOs

Large language models are no longer confined to research labs or academic benchmarks. Today, they operate as real-time services deployed at massive scale, serving millions of users across applications such as chat assistants, search engines, copilots, and enterprise tools. Unlike the training phase, which is compute-intensive but offline and non-interactive, inference happens live, often in response to a user prompt. This shift makes inference a fundamentally different challenge: it must deliver responses quickly, reliably, and at scale. A delay of even a few hundred milliseconds can significantly affect user experience, especially in interactive use cases like chat or code completion. At the same time, inference is resource-hungry. The underlying infrastructure relies on scarce and expensive hardware like GPUs, and these resources must be shared efficiently across thousands of concurrent requests. On top of that, workloads are highly dynamic. Traffic can spike unexpectedly, inputs vary in length and complexity, and different applications place different demands on the system. In this unpredictable environment, it becomes essential to define clear goals for what the system should prioritize. Service Level Objectives (SLOs) provide this guidance. They define acceptable performance boundaries and help teams make informed tradeoffs between latency, throughput, cost, and quality. Without them, optimization efforts risk being misaligned with real-world expectations, leading to either overprovisioned systems or poor user experience.

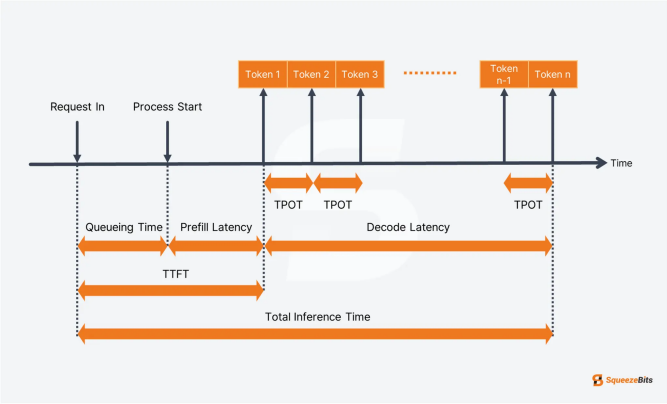


Figure 3.18: Breakdown of Inference latency that impacts SLO targets

Service Level Objectives (SLOs) are measurable targets that define the expected performance of a system. In the context of LLM inference, SLOs might specify that

the time to first token (TTFT) should be under 200 milliseconds for 95 percent of requests, or that the cost per thousand tokens should stay below a certain threshold. Other examples include time per output token (TPOT), request success rate, or GPU utilization efficiency. What distinguishes SLOs from simple metrics is intent. While metrics describe what is happening in the system, SLOs define what should happen. They set the performance boundaries within which the system is considered healthy. SLOs also differ from Service Level Agreements (SLAs), which are formal contracts between a service provider and a customer. SLAs are legal commitments and often carry financial penalties if not met, whereas SLOs are internal goals that guide system design, monitoring, and operational response.

Different stakeholders rely on SLOs for different reasons. Infrastructure engineers use them to optimize how resources like GPUs are allocated and batched. Site Reliability Engineers (SREs) use them to set up alerts and to decide when to trigger incident response processes. Product teams refer to SLOs to ensure that user-facing features deliver a smooth and responsive experience. In enterprise settings, SLOs also inform discussions with customers, particularly for API-based LLM services where consistent quality and performance are crucial. In this way, SLOs act as a contract between the system and its stakeholders, given they align engineering efforts with real-world expectations.

However, it is important to note that different applications of large language models place value on different performance goals, and as a result, each use case requires a distinct set of SLOs. For instance, systems designed for offline processing care far less about latency. In these settings, such as document classification/tagging or metadata generation, the priority is maximizing throughput in order to process as much data as possible at the lowest possible cost. These workloads often run in the background or overnight, where response time is less critical than total job completion time and GPU utilization.

By contrast, real-time systems such as chatbots or interactive agents care deeply about latency. A user waiting for a reply expects the first words to appear almost instantly. In these cases, time to first token (TTFT) is the most important SLO, because it directly shapes the user's perception of responsiveness. Once generation begins, time per output token (TPOT) only needs to be fast enough to keep up with the user's reading speed. This balance allows systems to save compute while still delivering a fluid experience. However, in use cases like LLM-powered agents, where the model must perform reasoning steps, call APIs, or trigger external tools, what matters most is end-to-end latency. These systems may wait on multi-step chains of thought or external calls, so the goal is to minimize total response time across all components.

An intuitive way to think about these tradeoffs is to consider the interaction style. When a human is actively waiting, as in chat or code assist, the system must respond quickly and smoothly. But when the task is happening in the background, like indexing or batch summarization, the focus shifts to how much work can be done per unit of time and cost. SLOs formalize these needs so that each system is tuned for what actually matters to the user or the business.

3.4.2 Parallelism Strategies

Meeting the right SLOs is not simply a matter of adding more compute or scaling blindly. Each objective such as latency, high throughput, or predictable cost places specific demands on how the underlying infrastructure is designed. These demands become especially critical when serving LLMs at scale, where the cost of inefficiencies can multiply quickly. In this context, parallelism strategies have become one of the most powerful levers available. Whether it is optimizing how tokens are generated in parallel, how requests are batched across GPUs, or how computation is distributed across nodes, the choice of parallelism strategy has a direct impact on a system's ability to meet its SLOs. The next section explores these strategies in detail, focusing on how different forms of parallelism can be used to navigate the tradeoffs between latency, throughput, and hardware efficiency.

Natural Language Processing (NLP) is advancing quickly in part due to an increase in available compute and dataset size. The abundance of compute and data enables training increasingly larger language models via self-supervised pretraining [50]. Empirical evidence indicates that larger language models are dramatically more useful for NLP tasks such as article completion, question answering, and natural language inference. As these models become larger, they exceed the memory limit of modern accelerators like GPUS. One solution has been proved to be to split the model across multiple of them. This not only alleviates the memory pressure, but also increases the amount of parallelism that can be done independently of the microbatch size, since multiple accelerators can process requests at the same time.

Withing the paralellism strategy, there are two paradignms: Data Parallelism and Model Parallelism.

Data Parallelism primarily involves deploying multiple replicas of the model on different GPUs. All model replicas are the same and share identical parameters. The key idea is that each replica independently processes user requests without needing to communicate intermediate computations with the others. Instead of having a single model instance process one input at a time, data parallelism allows the system to scale horizontally by assigning each incoming prompt to an

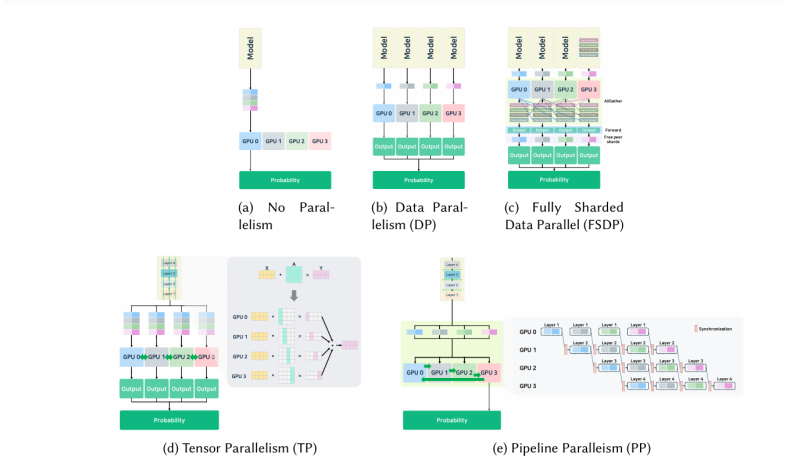


Fig. 12. Comparison of Parallelism Strategies

Figure 3.19: Comparison of Parallelism Strategies

available replica of the model. Each replica then performs the full forward pass for its assigned prompt or batch (collection) of prompts and returns the output independently. In production systems like OpenAI's ChatGPT, data parallelism is one of the core strategies used to serve massive volumes of inference requests. The model is deployed across hundreds or thousands of GPUs in the data center, and each group of GPUs, or nodes, runs its own standalone copy of the model, often referred as worker. In order to manage the large volume of requests per second, load balancing is used as a mechanism to route requests to available workers, aiming to distribute the workload evenly across available resources, avoid overloading any single GPU, reduce inference latency, and maximize system throughput. While data parallelism is effective for increasing throughput during inference by replicating the model across multiple GPUs and distributing input prompts among them, it falls short in one critical scenario: when the model itself is too large to fit into the memory of a single GPU. As language models grow in size, reaching tens or hundreds of billions of parameters, the memory footprint of a single model instance can easily exceed what is available on any one device. In such cases, data parallelism becomes insufficient, since each replica requires the full model to be loaded.

To overcome this limitation, large-scale systems employ model parallelism. The high-level idea behind model parallelism is to divide the model itself across multiple GPUs so that each GPU stores and computes only a part of it, which allow multiple GPUs to work together as a single logical unit to process a request end-to-end.

A common implementation of model parallelism is tensor parallelism, in which each layer of the model is computed collaboratively across multiple GPUs. By splitting the linear layers (which are essantially matrix multiplications) inside a transformer, such as attention heads, or feed forward layers, each GPU can handle a slice of the computation of each layer, rather than assigning entire layers to different GPUs. This enables multiple GPUs to collectively act as a single large GPU that hosts and executes the model.

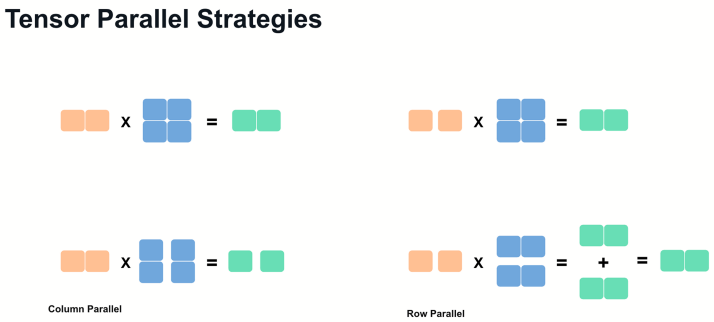


Figure 3.20: Column-wise vs Row-wise Tensor Parallelism

There are two main ways to split a tensor across GPUs in tensor parallelism: by rows or by columns. In the row-wise method, the weight matrix is divided along its output dimension, so each GPU holds a different set of rows. During computation, each GPU receives the full input and produces a partial output vector that corresponds to its assigned rows. Once all GPUs have completed their local computations, their outputs must be concatenated to form the full output of the layer.

In the column-wise method, the weight matrix is split along its input dimension, meaning each GPU stores a subset of the columns. The input vector is also partitioned so that each GPU receives only the portion relevant to its slice of the matrix. Each GPU computes a partial result, and these must be summed across all devices to obtain the final output.

Both approaches effectively parallelize the matrix multiplication, but they require that the GPUs exchange intermediate results at the end of each computation, which must be done for every layer of the model. That means cross-GPUs communication therefore needs to happen in order for the final output to be delivered. This communication step occurs once per layer and introduces an inherent latency cost, and the extent of this cost depends heavily on the speed and bandwidth of the interconnect between GPUs.

When fast links such as NVLink or NVSwitch are available, the overhead remains relatively low. On the other hand, in systems connected over slower buses like PCIe, communication delays can become a significant performance bottleneck. For this reason, the efficiency of tensor parallelism is closely tied to the underlying hardware and the design of the communication layer. However, it is important to note that Tensor parallelism cannot scale indefinitely. There is a practical ceiling, after which the cost of synchronization and the lack of meaningful computation per GPU outweigh the benefits of parallel compute, and diminishing returns or even worse performance can be detected.

Nontheless, despite the added communication overhead, tensor parallelism often reduces overall end-to-end latency by enabling multiple GPUs to work in parallel on each layer, which makes it particularly well-suited for real-time applications. For that reason, tensor parallelism remains one of the most effective strategies for enabling large-scale language model inference.

Another common implementation of model parallelism is pipeline parallelism. In this implementation, the layers of the neural network are divided into sequential stages, with each stage assigned to a different GPU. Instead of splitting the computation within individual layers, as in tensor parallelism, pipeline parallelism assigns full blocks of consecutive layers to different devices. It is primarily used when the model is too large to fit within the memory limits of a single GPU or even across a few GPUs using tensor parallelism alone. By distributing entire segments of the model across multiple GPUs, it allows very deep networks to be executed end to end. One of its main advantages is that it reduces memory pressure on individual GPUs and allows the model to scale across a larger number of devices without requiring each GPU to coordinate computation within layers. However, pipeline parallelism also comes with limitations. Because the input must pass through the pipeline stages one after another, there is an inherent delay before the output can be produced. This is especially problematic during inference with small batch sizes, where the lack of parallel work across tokens leads to idle time, known as "pipeline bubbles," in which some GPUs are waiting for others to finish their stage. As a result, pipeline parallelism can underutilize hardware and increase per-token latency unless the pipeline is kept full, typically through large batch sizes or micro-batching techniques.

In real modern LLM system, a combination of the three parallelism strategies are employed to achieve efficient and scalable performance. These strategies are not mutually exclusive but rather complementary. Typically, tensor parallelism is applied within a single machine, leveraging the high-bandwidth interconnects between GPUs to split the computation of individual layers. Pipeline parallelism is then used across machines or nodes in a cluster, with each node handling a different

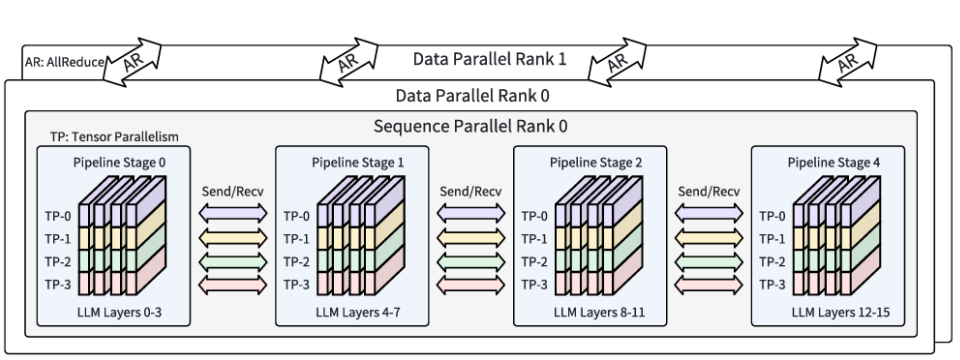


Figure 3.21: An example of 3D-parallelism with data-parallelism, tensor parallelism, and pipeline parallelism.

segment of the model. Finally, data parallelism operates at the outermost layer, replicating the full pipeline-tensor stack across multiple groups of GPUs to process different inputs in parallel and maximize system throughput.

3.4.3 The Quadratic Complexity in the Attention Mechanism

The attention mechanism is the core feature that made the transformer architecture so effective. By enabling each token to attend to every other token in a sequence, it captures subtle patterns, long-range dependencies, and nuanced context. This capability has played a central role in the success of large language models across a wide range of tasks. However, the same mechanism introduces a significant computational burden during inference, especially in autoregressive generation.

In autoregressive generation, a model produces one token at a time. With each new token, the model reprocesses the entire sequence generated so far. This involves tokenizing all previous words, embedding them, and computing the corresponding key (K) and value (V) vectors for each token. These steps are repeated at every generation step, even for tokens that have already been processed in earlier steps.

The attention block performs pairwise comparisons between every query (Q) and every key (K). For a sequence of n tokens, this results in an $n \times n$ attention matrix, where each position reflects the attention score between two tokens. As a result, the time and memory complexity of self-attention grows quadratically with sequence length. This behavior, described formally as $O(n^2)$, becomes inefficient as sequences grow longer, particularly during generation.

To illustrate this more clearly, consider the phrase:

"The quick brown fox jumps over the lazy dog."

Assuming this tokenizes into 9 tokens, generating the 9th token requires computing attention scores between it and the 8 that came before. For the model to do this, it must re-embed all previous tokens, recompute their K and V vectors, and then calculate the attention scores between the new query and each of those 8 keys. Now consider a sequence with 1,000 tokens. To generate token 1,001, the model must reprocess all 1,000 previous tokens. This process repeats for every additional token, which leads to rapidly increasing computation.

A crucial observation can be made here. While the query vector changes at each generation step, since it corresponds to the newly generated token, the keys and values of previous tokens do not change. Once the K and V vectors have been computed for earlier tokens, they remain valid and can be reused.

This idea lead to the creation of a computation optimization technique, called KV Caching. Instead of recomputing the keys and values at every step, the model can store them after their first computation and reuse them in subsequent steps. Only the new query vector needs to be calculated and compared against the cached keys. This eliminates redundant computation and reduces the cost of attention from quadratic to approximately linear with respect to sequence length.

Such an optimization has become essential in large-scale deployments of language models, where improving throughput and minimizing latency directly translates to better performance and lower cost. KV caching allows for significantly faster generation without affecting output quality, and has become a standard component in nearly all production-grade inference systems.

3.5 KV Caching and the Rise of Memory Bottlenecks

3.5.1 KV Caching

The following section explores the KV caching technique in greater detail. It covers how the cache is implemented and how it affects memory and performance. Most importantly, this section discusses the new challenges KV Caching introduces when scaling LLMs across machines or GPUs.

KV caching represents a deliberate tradeoff. In autoregressive models, computing the key and value representations for all previously generated tokens at each decoding step leads to a computational cost that scales quadratically with sequence

length. This cost becomes prohibitive in real-world workloads. To mitigate this issue, KV pairs corresponding to past tokens are stored in GPU memory, avoiding redundant computation during each forward pass [51].

However, this shift from compute-bound to memory-bound workloads introduces a new set of constraints. The principal concern is that GPU memory is both expensive and limited. As described in section 3.3.2, a modern high-end GPU typically provides around 90 GB of usable memory. As will be shown in this section, this capacity is insufficient to support the thousands of concurrent requests required in large-scale inference scenarios. This section will uncover how such memory capacity is insufficient to accommodate the thousands of concurrent requests typically required by large-scale inference workloads.

Per-token memory consumption (in bytes) of the KV cache $= 2 \cdot n_{\text{layers}} \cdot n_{\text{heads}} \cdot d_{\text{head}} \cdot P_a$

o understand the extent to which KV caching can become a bottleneck, it is useful to compute the amount of memory required per token. The first step is to realize that for each new token that the model generates, both the K an V need to be stored. The Q does not need to be stored because it is only used one time to compute the attention between the current token and the previous ones, meaning previous tokens do not need to store their Q value. However, each of these Ks and Vs arent just a single number, instead, they are a vector of dimension d_{head}. How big d_{head} is will depend on the model itself, and it is a reflection of how many degrees of freedom the model has to compute attention, meaning that the more dimensions a vector has, the more contextual meaning we can infuse into the vectors, as previously mentioned with the "second" example in section 3.2.2. The size d_{head} is also normally related to the number of parameters a model has. The larger the model is, the higher d_{head} tends to be, which is also why they allow for more contextual understanding and why larger models tend to perform better than larger ones (but point out exceptions). Since K and V are vectors of dimensiond_{head}, they are a composition of individual numbers, also called parameters. Each number is represented can also be represented in different precisions. A good metaphor would be how we can refer the number pi to be 3.1415926535, 3.1415 or 3. The less precision is used, the less memory it occupies to store, but also less precise it is. Similarly, in LLMs, each of the numbers in a tensor can be stored with FP32 (32 bits) , which is equivalent to 4bytes/parameter, FP16 (16 bits) which is equivalent to 2bytes/parameters, FP8 (8bits) which is equivalent to 1byte/parameter . Other types like BF16 also exists, but it still occupies 2bytes/parameter. Like in the pi case, the larger the precision is, the more accurate outputs and better responser the model will be able to deliver, but it will come at the cost of needing to store in more memory, which is very costly. By reducing the precision for example

from FP16 to FP8, the model parameter memory occupation drops to half the original, which for big state of the art LLM models used for inference, typically ranging from dozens to hundreds of billion of parameters, is a huge memory saving. Another thing to take into account when calculating the memory cost of a single K V pair for a single token, is the fact that most transformer architectures now have multiple attention heads per layer to be able to capture different rich contextual meanings and nuances at the same time, and each one has to have its own set of Ks and Vs [39]. This means that for a single token, the memory utilization of a K and V pair needs to be multiplied by the number of attention heads that exist, referred to as n_{heads}. Whats more, the transformer architecture has multiple layers n_{layers}, so it also has to be multiplied by the total amount of layers. This total KV Cache memory utilization is just for storing a single token. In real life scenarios, users will use multiple tokens, that can be broken down into two categories: the prefill, which is the input tokens that the user gives to the model as a prompt for the request; and the decode tokens, which are the output tokens the model gave the user as an answer [52]. Therefore means that the total memory consumption of a single user, in reality, is not only the memory consumption of a single token, but the sum of memory consuption for each of the tokens at both the prefill and decode stages of inference. This is one of the largest current problems in inference systems, because memory consumption is therefore dependent of the sequence length of a users prompt and its answer, yet these values cannot be known ahead of time. The number of tokens a user will use for its request/prompt can not be known before the request is made, and the length of the text that the model will output is also not known a priori. This makes it really challenging to optimize memory, because systems need to account for varying length workloads in real time, without being able to know how much memory each user requires ahead of time. The problem magnifies at large scales, when thousands of requests need to be processed in parallel, each having its own sequence length, which is not known a priori.

3.5.2 Memory Optimization Techniques

The previous section showed that the key-value cache is both the reason large language models can generate quickly and the reason they so easily run out of GPU memory. Once every layer starts appending its new keys and values, the cache grows in a straight line with sequence length, and it must stay resident for as long as the conversation lasts. In practice this turns memory, rather than arithmetic throughput, into the true limiter of contemporary inference. That observation has guided an intense sequence of engineering efforts whose common goal is to optimize the utilization of memory during inference, leading to higher proportion of SLO achievements.

In the very first wave of public large-language-model demos, like GPT-2 in 2019, researchers tried doing inference directly on deep-learning frameworks like Pytorch [53]. Given that during inference only the forward pass needs to be computed, instead of additionally computing the backward pass, in which gradients are required, these parts that track them can be disabled to be optimized for inference. This "raw-eager" recipe worked because early open-source LLMs were only a few hundred million parameters and prompts were short. As soon as users asked for longer contexts, two structural problems appeared. The first one was due to the fact that concurrent requests coming from multiple users were grouped together, so that all these requests could be sent to the model in one shot. This strategy of grouping multiple user requests is called batching, and it lets the hardware be able to compute multiple requests in parallel, effectively increasing throughput. However, this strategy did not take into account that requests from different users can diverge hugely on length. For example, one user might request a short reply while another pushed the model toward a multi-paragraph essay. This became a problem, because since very item in the batch had to march through the network layer by layer in lock-step, the quickest request could not be returned to its owner until the slowest request had reached the same layer. Requests that have finished earlier than other requests in a batch cannot return to the client, while newly arrived requests have to wait until the current batch completely finishes. The short sequence therefore stalled inside the GPU even though it no longer needed any computation. This resulted in having a noticeable share of GPU capacity idle at every forward pass.

Inspired by this issue, researchers at the univerity of Seul introduced a scheduling mechanism that schedules execution of concurrent requests at the granularity of iteration (instead of request) [54]. With this approach, the scheduler now receives an output on every iteration, which corresponds to the generation of one single response token, and it can therefore detect the completion of a requests and immediately return its generated token to the client, without making it wait for other requests to be finished. It also means that once a request is finished, another one can be started in the next iteration, not having to wait in the queue until the whole batch is finished.

This was a big step in optimizing inference. However, another obstacle that was still highly relevant in these systems appeared in memory. As discussed in previous sections, for every new output token that the model generated as part of its response, is Key and Value would have to be stored in memory to reduce the computational overhead for next tokens. However, previous LLM serving systems fell short in optimizing this KV Cache memory allocation. They worked like this: Each time a new request was processed, they would reserve a fixed and contiguous

amount of memory space in the GPU, that it would correspond to the set amount of output tokens that the model answer was limited to (eg: 2048 tokens). However, the challenge is that in inference, the amount of space it will take, as well as its lifetime, cannot be known a priori. A very short output will therefore not fully occupy the reserved memory space it was given, and a too large output will see its answer cut short because it has ran out of its reserved chunk of memory. This phenomenon of under-utilizing reserved memory is called internal fragmentation, and it used to account to about 57.3% of unused GPU memory [55]. In real life, where GPU memory is both scarce and extremely expensive, this inefficiency became a huge problem, especially for large scale systems. Another type of fragmentation was the external fragmentation, that happened because the system could have many different maximum sizes for the reserved memory. Over time, smaller blocks that were being liberated after the end of their conversations, left small gaps between other larger blocks. This meant that even if the memory was liberated, another large memory block of new KV Caches could not be put inside these blocks, because it could not fit properly in the tiny space. Since splitting the KV values in a non-contiguous manner was not possible, plenty of gaps were left unused. The total amount of free space might be large, but because it is broken into many separate pockets rather than one continuous stretch, a program that asks for a single big block cannot be satisfied even though, in theory, enough bytes are available. It is like a parking lot where many narrow gaps sit between parked cars; all the gaps together could fit a bus, yet the bus cannot enter because no single gap is long enough.

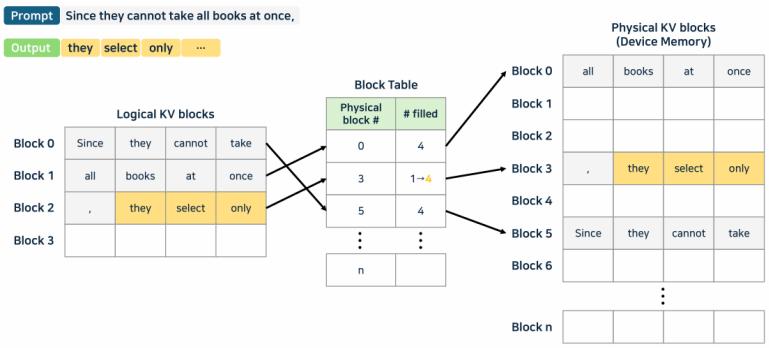


Figure 3.22: Paged Attention

Researchers from UC Berkley and Standford found a way to solve this problem.

They took inspiration on the way that operating systems (OS) handle allocation and management of memory in an efficient way, and applied it to LLM inference systems. The technique, called PagedAttention, is based on the realization that having big memory blocks that have to be stored in a contiguous way to store KV Cache can lead to both internal and external fragmentation [55]. In order to mitigate them, user requests are instead broken down into fixed-size pages, that represent small chunks of memory. One request's KV pairs are composed of a sequence of contiguous logical pages, but these logical pages are then mapped to different physical memory locations, that do not have to be physically stored in a contiguous manner inside the GPU. This is done by creating a block table, that maps where is each logical KV block/page stored in the physical KV pages. TO illustrate this, when a request is first process, the algorithm will start to store each token's KV pairs in one block/page, until no space is left. Then, for the next token, another block is allocated for the new KV pair to be stored, which does not have to be physically contiguous to the previous block, and this happens until the EOS token is generated, or the maximum output length is reached. By storing the KV values from one request in many logical small blocks, which are all the same size, the internal fragmentation is greatly reduced, because the maximum wasted memory space for each request will be equivalent to the size of the page, which is very small compared to the single block where the whole request's KV pairs were previously stored. Additionally, external fragmentation, that is, those gaps of unutilized memory that stood between requests, are completely eliminated, since all the percentage of GPU memory that is reserved for KV values is now divided in same-length pages or blocks, so there is no wasted gaps between each one.

In summary, PagedAttention algorithm allows KV blocks to be stored in a non contiguous manner, which fully eliminated external fragmentation in GPU memory while greatly reducing internal fragmentation.

Since the debut of PagedAttention, several other branches of research have refined or extended this idea. PagedAttention brought significant memory savings by allocating GPU memory on demand, but it came at a cost. It broke the illusion of a clean, contiguous memory layout, forcing attention kernels to handle fragmented memory manually. This meant that attention kernels could no longer access the key-value (KV) cache with a simple linear pointer. Now, before the kernel can read token T's key or value, it must consult a page table (a mapping structure) to figure out where in physical memory the block for token T is located. This lookup step added a layer of complexity because the kernel could not assume simple memory layout anymore. It also increased code implementation complexity, and introduced a small but persistent performance penalty during inference. Researchers at Microsoft realized these problems and came up with vAttention, a new

approach that preserves the simple, contiguous view of memory expected by high-performance attention kernels, while still allocating physical memory only when needed [56]. Instead of relying on the application page tables and manual memory management, vAttention leverages CUDA's virtual memory system to reserve a long, contiguous range of virtual addresses at the start of each request. Physical memory is then mapped into that virtual range on demand as the conversation grows. Because the kernel sees a clean, unbroken address space, no special logic is needed to dereference tokens, and unmodified kernels like FlashAttention-2 or FlashAttention-3 can be used directly [57]. This design removed the need for extra lookups in the code and lets the model run faster, while still keeping memory usage efficient and avoiding unnecessary waste.

Recently, there has been a wave of new optimization techniques that go beyond local memory management and explore broader system-level strategies for scaling LLM inference. These approaches are often driven by the realization that the attention mechanism, and especially the KV cache, is the main factor limiting memory capacity. One example is Infinite-LLM, which builds on the idea that attention layers consume the most memory, while other parts of the model such as MLPs and normalization layers are relatively lightweight [58]. Based on this, the system offloads attention computations and KV cache storage to a set of secondary GPUs, allowing the main GPUs to handle the rest of the network. This makes it possible to support much longer sequences without overwhelming a single device. Another example is Mooncake, which is grounded in the belief that not all inference tasks need to happen on the same machine [59]. It introduces a disaggregated setup where the prefill and decoding stages are handled by different GPU groups, and the KV cache is shared across a cluster-wide pool that includes both CPU memory and local storage. A dedicated scheduler manages what gets loaded or evicted, helping the system handle a larger number of simultaneous conversations than would be possible in a traditional single-node setup.

3.5.3 Inference Engines

The recent rapid expansion of large language models has resulted in a wide range of services powered by artificial intelligence, such as chatbots, virtual assistants, code-generation tools, and AI-augmented search engines. These models rely on a common architecture and are typically trained on massive datasets using large-scale distributed infrastructure. As previously discussed, during training, the primary objective is to teach the model how to predict the next token given a context. This workload is highly predictible as the process always involves preparing batches of uniform data, pad them to the same length and process them in parallel across thousands of GPUs.

Inference, on the other hand, is a different story. It happens in the real world, which means that rather than running on fixed-size batches, the model must now respond to a live stream of requests that vary widely in length, complexity, and urgency. For example, some users might send a short question and expect an answer in under a second, while others might request a full essay or code generation that spans hundreds of tokens. Each request must be processed as it arrives, and often begins generating one token at a time, which is far less efficient for GPU compute. Moreover, unlike training, which can be paused and restarted in a controlled environment, inference systems must run continuously and remain responsive under unpredictable workloads.

This shift from training to inference introduces a new set of constraints and bottlenecks that were not present during training. The system must now make real-time decisions about how to schedule requests, allocate memory dynamically, and prevent compute or memory resources from sitting idle.

The memory optimization techniques such as PagedAttention, parallelism techniques such as Tensor and Pipeline Parallelism, and batching techniques are ways to improve the efficiency of serving systems [55]. These techniques are valuable on their own, but they only reach their full potential when guided by a system that knows how and when to use them. Optimizing one component in isolation can bring measurable improvements, but serving large language models at scale requires balancing many moving parts at once that all are interconnected, and any one of them can become a bottleneck if the others are not coordinated properly.

In order to solve this issue, specialized systems called inference engines have emerged [60][61][62]. Inference engines represent a coordination layer that sits between the model and the hardware, and is responsible for making decisions on the fly about how to group requests, how much memory to allocate, and how to keep the GPUs busy without creating delays. They utilize multiple optimization techniques discussed in Section 3.5.2, in a way that adapts continuously to changing workloads and system conditions.

Chapter 4

Project Definition

This chapter aims to justify the development of the project by addressing the motivation behind it. Additionally, it outlines the objectives to be pursued, describes the methodology employed, and details the project's schedule and budget estimate.

4.1 Motivation

The past two years have witnessed an unprecedented surge in the use of generative-AI services. Chatbots, code companions, and content-creation platforms now attract a volume of queries that would have seemed implausible only a few product cycles ago. What began with a few thousand simultaneous conversations per model has expanded to tens of thousands of concurrent sessions sustained throughout the day. Major cloud providers already report petabyte-scale inference workloads, a figure that continues to rise as new applications emerge and user bases expand geographically.

This rapid adoption reshapes expectations for responsiveness. End users perceive conversational AI as an always-on utility and demand replies that arrive in real time, much like the response of a search engine or a messaging app. Traffic spikes triggered by global news events, marketing campaigns, or software releases can multiply baseline demand within minutes.

Scaling these services is not merely a question of purchasing additional hardware. A single NVIDIA H100 accelerator costs about USD 25 000 and draws more than 700 W under load. Production clusters often deploy thousands of such GPUs, bringing capital expenditure into the hundreds of millions of dollars before accounting for networking, cooling, and physical space. Operating expenses add

even more weight to the bill. The electricity required to run the GPUs, ongoing maintenance contracts, and regular hardware refresh cycles can push the lifetime cost of ownership into the billion-dollar range for a large-scale provider. In this economic context, the industry has shifted its focus away from continually acquiring costly hardware toward maximising the utilisation of the infrastructure already in place, since each percentage of under-utilised translates to a direct financial penalty.

The industry priority is therefore to extract the maximum performance from the existing infrastructure. However, achieving this goal calls for a rigorous, data-driven grasp of how model design, workload characteristics, and hardware constraints interact under real operating conditions. Only by mapping these relationships in detail can engineers devise scheduling, batching, and precision strategies that keep latency low while pushing memory and compute resources to their practical limits. This project sets out to provide that empirical foundation, delivering the insights needed to scale generative-AI services responsibly and cost-effectively.

4.2 Objectives

The main objective of this project was to conduct a multivariable analysis of LLM inference performance at scale of the underlying AI infrastructure, in order to identify the factors that most drive resource inefficiency and, in turn, worsen the cost structure for AI infrastructure providers. To enable this analysis, a second objective was to design and implement a reproducible benchmarking pipeline that deploys open-source language models on an H100 cluster, generates realistic streams of user requests, and captures fine-grained hardware telemetry in real time so that performance can be characterised across workloads. Finally, a third objective was to distil the empirical findings into insights and recommendations for operating LLM inference services more efficiently. The remainder of this section explains these objectives in greater detail:

- **Conduct a Multivariable analysis of large-scale LLM inference:** Quantify how inference performance and resource efficiency change when key workload and system factors vary, and identify the main drivers of resource underutilisation that deteriorate the cost structure of infrastructure providers.
- **Design a reproducible benchmarking pipeline:** Design and implement an automated and reproducible benchmarking pipeline built to deploy open-source large language models on an H100 cluster, generate synthetic streams

of user workloads, and record fine-grained hardware metrics at sub-second resolution, in order to produce the datasets required for the multivariable analysis.

- **Deliver role-aware operating guidance grounded in empirical results:** Convert the benchmark results into data-driven recommendations that span the full stack of roles, from developers and system architects to infrastructure providers and hardware manufacturers, enabling each to apply the findings to make better choices and improve the efficiency of the AI ecosystem.

4.3 Methology

An agile methodology will be adopted for the development of the project, founded on iterative and incremental software production. The work will be organised into successive sprints, each defined as a short, fixed period. Within every sprint, tasks will be identified, prioritised, and addressed in a cooperative and flexible manner.

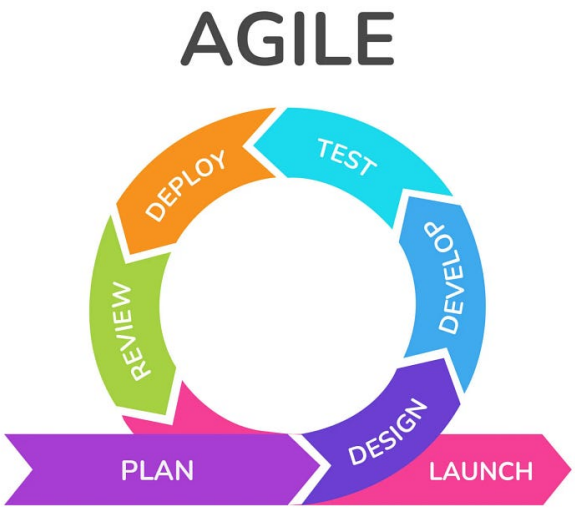


Figure 4.1: Agile Development Cycle

The intention is that tangible results will be produced during each iteration, so that continuous adaptation can be achieved as the project advances. By this means, feedback will be gathered regularly, and adjustments or improvements

will be introduced whenever new needs or requirements are recognised during the development process.

The agile methodology was selected because of the wide range of advantages it offers over other software-development approaches. The main advantage is that productivity is raised by organising the work into sprints, which makes rapid and frequent deliveries of functional code possible. This practice keeps the product in a usable state at all times, allowing feedback to be gathered from users and adjustments or improvements to be introduced.

The approach also encourages members to collaborate continuously, which helps everyone make faster, better decisions. Altogether, these benefits lead to higher-quality software and allow the project goals to be met more efficiently.

4.4 Planning

This section outlines the planning methodology adopted for this project. It explains how the work was divided into well-defined tasks and distributed across the entirety of the alloted time.

Task \ Month	**Oct**	**Nov**	**Dec**	**Jan**	**Feb**	**Mar**	**Apr**	**May**
Project kick-off & scope lock-in	✓							
Deep literature review & problem refinement		✓	✓	✓				
Benchmark Suite Design & Selection				✓				
Development and Deployment Metrics-Collection Subsystem					✓			
Implementation of Synthetic User-Traffic Generation Subsystem					✓			
Implementation of Automated Orchestration Layer						✓		
Execution of Benchmarks						✓	✓	
Analysis and Validation of Experimental Results						✓	✓	
Thesis Writing		✓	✓	✓	✓	✓	✓	✓

Table 4.1: High-level project schedule.

The overall timeline is presented in Table 4.1. Each row indicates the months allocated to a specific task, with overlaps introduced to shorten feedback loops and minimise idle time. The execution and results of all tasks shown in Table 4.1 are discussed in detail in Chapter 5.

Operationally, task tracking was handled with Trello. A single board held three workstreams (thesis documentation, system implementation, and benchmark execution and analysis) which were further broken down into cards. Cards flowed through the standard To Do, In Progress, and Done columns, providing an up-to-date snapshot of progress and helping to prioritise daily effort.

4.5 Economic Study

This section develops an economic analysis of costs, revenue, and profit from the perspective of a core stakeholder in the ecosystem: the infrastructure provider. The objective is to translate the behavior observed under controlled LLM inference workloads into a concise model that informs capacity planning, pricing, and operational targets.

Drawing on the experimental results reported in Section 6, a common economic model emerges as a conceptual abstraction of inference system behavior. The model holds regardless of the specific LLM, hardware vendor, or deployment configuration, provided that usage is priced per unit of useful work and throughput is taken as the measure of output.

Figure 4.2 presents the Infrastructure Performance Profile, plotting aggregate throughput, understood as the useful work delivered by the entire inference system, as concurrent user demand increases.

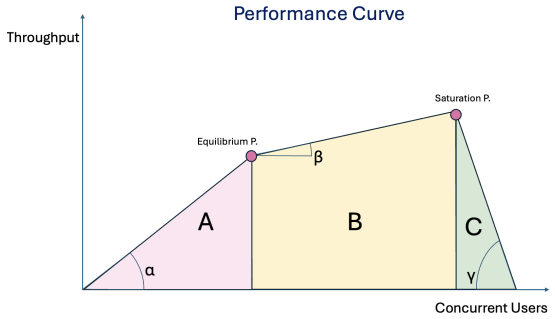
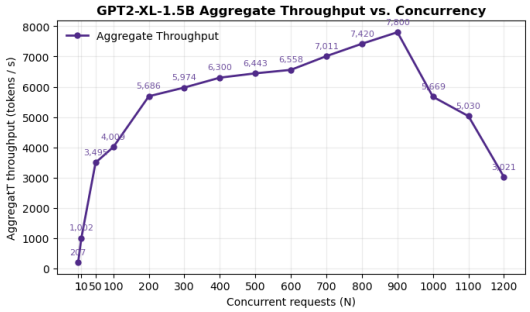
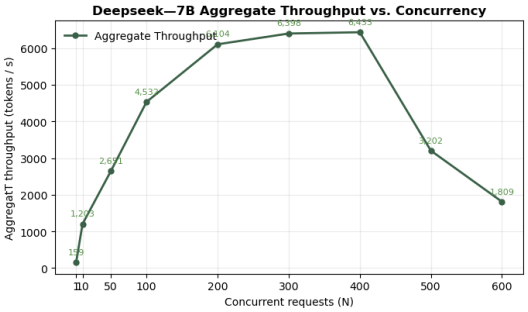


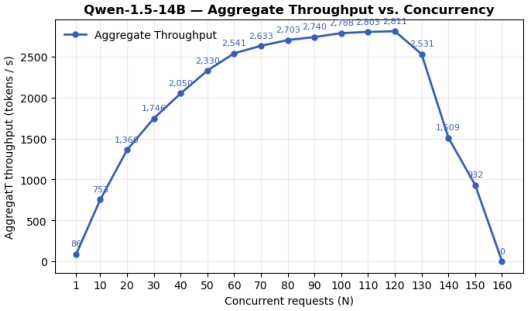
Figure 4.2: Infrastructure Performance Profile



(a) Concurrency threshold for KV-cache saturation (GPT2-xl-1.5B)



(b) Concurrency threshold for KV-cache saturation (Deepseek-7B)



(c) Concurrency threshold for KV-cache saturation (Qwen1.5-14B)

Figure 4.3: Concurrency threshold for KV-cache saturation as a function of model size across different models.

This conceptual abstraction of the LLM infrastructure performance profile shown in Figure 4.2, derived directly from the project's benchmark results such as the one shown in Figure 6.48, will serve throughout the economic analysis as a unifying abstraction, removing model- and configuration-specific details because the governing relationship remains invariant.

As shown in Figure 4.2, the Infrastructure Performance Profile delineates three operating regions A, B, and C, each with distinct characteristics that shape inference system behavior. These regions are:

Region A: Spare Capacity. In Region A the system operates with spare capacity, because provisioned resources exceed concurrent demand, which means work does not contend and queues do not form. As a result, each additional user translates almost one to one into useful work, so throughput scales in step with demand. As observed in the experiments reported in Section 6.1.2, the slope angle

α is less than or equal to 45 degrees.

$$\alpha \leq 45^{\circ}$$

Latency remains minimal, since requests begin service immediately and do not need to share scarce resources, and user experience reflects this absence of waiting. This regime ends at the "equilibrium point", where all resources become active, and beyond which the system transitions to Region B.

Region B: Shared Capacity and Emerging Queues. In Region B the system operates near full utilization, because demand begins to exceed the number of independent service slots, which forces resources to be shared across requests. One important characteristic of this region is that throughput still continues to increase, although with a smaller slope than in Region A, since part of the capacity is now spent on the overheads of sharing resources. This pattern is consistent with the queue formation documented in Section 6.1.2.

$$\beta < \alpha$$

Region B ends at the "saturation point". Beyond this point, the aggregate throughput declines, because the resources required to accommodate an additional request exceed the net useful work contributed by the system.

Region C: Overload and Throughput Collapse. In Region C demand exceeds the sustainable service capacity by a wide margin, which forces intense contention and coordination overheads. Useful work declines because shared resources are repeatedly reallocated, caches are thrashed, and waiting amplifies across the pipeline. The result is that each additional concurrent request consumes more capacity than the net work it contributes, so aggregate throughput falls while latency and error risk rise.

Building on the characterization of Regions A, B, and C above, the following summary consolidates the main trade offs and indicates, for each regime, whether conditions favor users and providers:

- **Region A** — Optimal for a newly arriving user because added latency is minimal, yet economically suboptimal for the provider since part of the capacity remains unutilized and revenue per unit of capacity is low.
- **Region B** — Resources are shared, which introduces additional waiting and slightly degrades user experience, while it is economically preferable for

the provider because aggregate throughput and utilization rise up to the "saturation point".

- **Region C** — Demand exceeds sustainable capacity, therefore each extra request reduces net useful work, so aggregate throughput declines and latency escalates, which makes this regime undesirable for both users and provider.

4.5.1 Cost Structure of Infrastructure Providers

Once the performance profile of inference systems is established, it becomes necessary to define the cost structure that limit infrastructure providers, since this structure sets the boundaries for pricing and the economically desirable operating region.

Provider costs can be grouped into two categories: fixed costs and variable costs.

Fixed Costs

On the one hand, fixed costs are incurred by infrastructure providers regardless of user workload. The following figure summarizes the main categories:

Fixed cost category	**Examples**
Hardware acquisition and depreciation	GPUs or TPUs, servers
Rack and power delivery	Racks, PDUs or UPS
Networking fabric and optics	Switches, structured cabling
Datacenter space and facilities	Colocation or lease
Salaried staffing	SRE or platform engineers, facilities staff
Physical security and compliance	Access control, guard services
Insurance, taxes, and audits	Insurance premiums, property taxes
Enterprise software and support	Orchestration or monitoring licenses, support

Table 4.2: Representative fixed cost components for infrastructure providers.

Variable Costs

On the other hand, variable costs scale with system usage. Some examples include electricity and water consumption, which rise with demand. Table 4.3 lists the most prominent variable cost categories for inference providers:

Variable cost category	Main drivers and examples
Compute electricity	GPU power, CPU power
Cooling energy and water	Chiller energy, evaporative water
Metered platform services	Load balancer usage, API gateway requests
Per request licensing	Model licenses, runtime licenses
Storage and data movement	Object storage, retrieval egress
Burst staffing and support	Overtime, contractor hours

Table 4.3: Representative variable cost components for infrastructure providers.

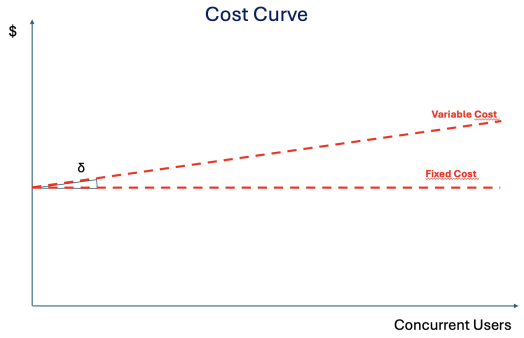


Figure 4.4: Total Cost Structure for Infrastructure Providers

Therefore, the total cost for infrastructure providers is the sum of fixed and variable components:

$$\text{Total Cost} = \text{FC} + \text{VC}$$

$$\text{FC} = \sum_{i=1}^{N_F} F_i \qquad \text{and} \qquad \text{VC} = \sum_{j=1}^{N_V} V_j$$

4.5.2 Revenue Structure for Infrastructure Providers

Revenue refers to the monetary inflow obtained by charging per unit of useful work, independent of the costs incurred.

For infrastructure providers, revenue follows the Infrastructure Performance Profile in Figure 4.2, because the service is priced per unit of useful work, typically

measured in tokens. As throughput rises, billable tokens increase and revenue grows.

Consequently, revenue inherits the three operating regions discussed earlier: in Region A it grows almost one to one with demand, in Region B it continues to rise with diminishing marginal gains due to resource sharing, and beyond saturation in Region C it declines as aggregate throughput falls.

Here is an example of how revenue would be computed in the case of infrastructure providers:

Let p_{1k} be the price per 1,000 tokens. The per-token price is

$$p_{\text{tok}} = \frac{p_{1k}}{1000}.$$

If $T_b(D)$ denotes billable tokens per second at demand D, then the instantaneous revenue rate is

$$R(D) = p_{\text{tok}}\, T_b(D) = \frac{p_{1k}}{1000}\, T_b(D).$$

Over an interval of length Δt, the revenue is

$$\text{Revenue}(\Delta t, D) \approx \frac{p_{1k}}{1000}\, T_b(D)\, \Delta t.$$

4.5.3 Profit Structure for Infrastructure Providers

Profit is defined as the difference between revenue and total cost:

$$\Pi = R - C_{\text{total}}.$$

Based on the cost and revenue structures defined in previous sections, the final profit structure takes the following shape for infrastructure providers, shown in figure 4.5:

Based on this profit analysis, Region A is undesirable for the provider because a portion of capacity remains idle, which implies overprovisioning and leaves fixed costs dominant. Consequently, Region A is both technically inefficient and economically unfavorable.

Moving into Region B, all provisioned resources come into play, which makes the regime at least technically efficient. Profit can still be negative at first, because fixed costs weigh heavily, but as demand rises a threshold is reached where revenue equals the sum of fixed and variable costs, the break even point. From that moment up to the "saturation point", each increment of demand produces positive profit

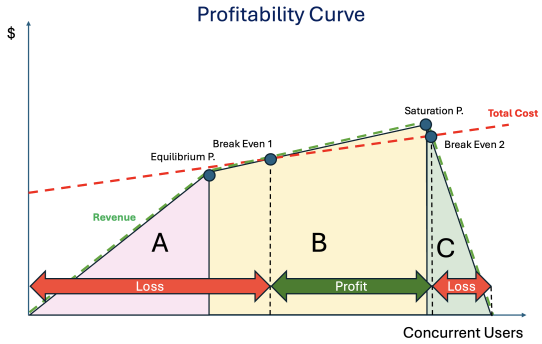


Figure 4.5: Profit Structure for Infrastructure Providers

and better utilization, and although gains diminish as resources are shared more intensely, the aggregate profit still benefits from more user load.

Crossing into Region C, the marginal request consumes more capacity than the useful work it adds, but variable costs continue to accumulate, so profit erodes until it can turn negative again.

Taken together, this economic study supports a clear operating objective: keep the system near the upper end of Region B, the saturation point, where profit is typically highest while service remains stable. To achieve this, providers should enforce per node admission control that keeps observed concurrency close to this point and scale horizontally to track demand, so individual nodes remain near the optimum rather than drifting into overload.

This guidance is stated without any fix numeric values, in order to remain model and configuration agnostic, allowing any provider to apply it regardless of deployment specifics.

However, to ground the guidance with concrete examples, Table 4.4 reports the per node concurrency at which aggregate throughput peaked for representative models, using the hardware configuration described in Section 5.1.1 and a comparable workload pattern. These values are derived from Figure 6.48, and for these examples, they represent the exact point where maximum profit can be obtained.

Model	Parameters	Per node concurrency at Saturation P.
GPT2-xl	1.5B	≈ 900
Deepseek-7B	7B	≈ 400
Qwen1.5-14B	14B	≈ 120

Table 4.4: Illustrative operating points at saturation under this study's benchmark conditions.

Since the saturation point shifts with model updates, hardware changes, and workload composition, it should be remeasured whenever those factors change.

Chapter 5

Implemented System

In this chapter, the implemented system will be described in depth. The platform has been organised as a modular pipeline whose subsystems cooperate to execute every benchmark automatically and reproducibly. Each subsystem is defined by a single, precise responsibility and can be invoked on its own for debugging or extension. When combined, the entire pipeline can be launched with a single command, performing a full end-to-end benchmark run without manual intervention.

To better understand the structure and function of the system, this chapter is divided into five main sections, each corresponding to a distinct phase of the implemented system or pipeline stage:

- **Benchmark Selection**: This section will describe the process of choosing models and tasks that reflect real-world large-scale LLM inference scenarios. The benchmarks that were finally selected span a diverse range of model sizes and deployment settings, in order to capture the performance characteristics and trade-offs relevant to modern AI workloads.
- **Automated Pipeline Orchestration**: This section will describe how the different subsystems were coordinated through automation scripts and configuration logic. The orchestration was meant to enable seamless execution of end-to-end benchmark run and also ensure that each component operated in the correct order with minimal manual intervention.
- **User Workload Generation**: This section will explain how the system simulated realistic user traffic patterns under varying load conditions. The goal was to evaluate how each model behaved under different user workloads, by generating inference requests that mimic real-world user usage scenarios, allowing a deeper understanding of performance and resource usage for each

case.

- **Hardware Telemetry Collection**: This section will explain what hardware-level data was captured in real time during each benchmark run, focusing on GPU and system resource utilization. It will also detail how these metrics were collected and stored in a way that enabled a clear view of the underlying infrastructure's behavior.
- **Metric Analysis and Visualization**: This section will explain how the collected hardware metrics were processed to better understand system performance. It will describe the scripts used to extract and filter the data, as well as how the results were visualized to compare workloads and highlight performance trends.

5.1 Benchmark Selection

The goal throughout this selection process was to create a representative and diverse set of experiments that reflect how large language models actually perform in practice. These benchmarks form the foundation for the analysis presented in Chapter 6, where their results are studied in detail to uncover insights about throughput, latency, memory saturation, and system efficiency.

To capture the behaviour of vLLM across a representative spread of real-world workloads, six open-source LLMs were chosen. The set ranges from a lightweight 1.5 B-parameter model baseline to a 70 B-parameter FP8 model, covering three size "buckets" (small, medium, large) and including both dense and long-context variants. Each model brings a distinctive property, that helps isolate different performance bottlenecks during inference.

Size Bucket	Model (Huggingface Repository)	Parameters	Architecture	VRAM (GiB)	Context Window (tokens)
Small	gpt2-xl	1.5 B	Decoder-only Transformer	2.97	1 024
Small	Mistral-7B-128k	7 B	Decoder-only Transformer	13.56	128 k
Medium	deepseek-ai/deepseek-llm-7b-chat	7 B	Decoder-only Transformer	12.87	4 096
Medium	Qwen/Qwen1.5-14B-Chat	14 B	Decoder-only Transformer	26.43	32 768
Large	tiiuae/falcon-40b	40 B	Decoder-only Transformer	76.93	2 048
Large	RedHatAI/Meta-Llama-3.1-70B-Instruct-FP8	70 B	Decoder-only Transformer	67.70	128 k

Table 5.1: Overview of selected models used for benchmarking

The set of experiments has been organised into three benchmark families, each chosen to highlight a distinct factor that influences the efficiency of large language-model inference. Grouping the tests in this way makes it possible to study factors such as workload concurrency, sequence length or intra-node parallelism without letting one variable obscure the others. The idea is that by isolating these angles, it is possible to uncover where bottlenecks originate and how they interact with model size and hardware configuration.

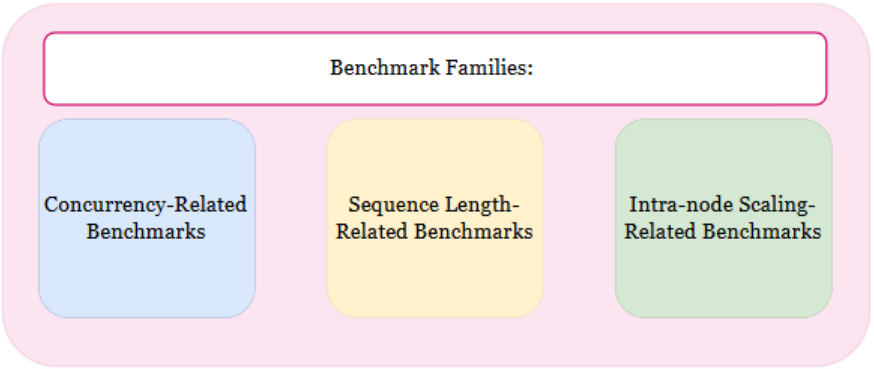


Figure 5.1: Benchmark Families

The following subsections introduce each family in turn, outlining its purpose, experimental setup, and the primary metrics of interest.

5.1.1 Concurrency-Related Benchmarks

Concurrency denotes the number of user requests the model must serve at once. A useful analogy is a single-barista café. When five people place their orders together, the barista prepares each drink in turn; as the queue lengthens, every customer's wait extends and the barista's workload rises. Similarly, when many requests arrive simultaneously at an LLM endpoint, the system must attend to them at the same time, which can slow individual responses and reduce overall efficiency.

In real-world LLM deployments concurrency stands out as a principal driver of performance, because production chatbots, retrieval-augmented-generation pipelines and code-completion services almost never process a single request in isolation. On the contrary, these systems routinely need to process hundreds or even thousands of queries each second while still being expected to satisfy tight service-level objectives for latency.

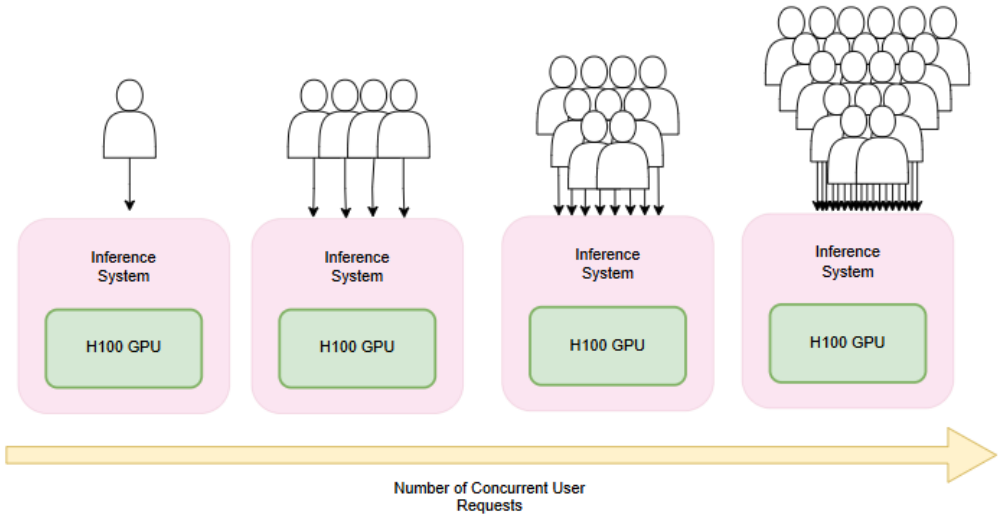


Figure 5.2: Concurrency-Sweep Setup: Increasing Numbers of Simultaneous User Requests Served by the Inference System

With this motivation in mind, the objectives of this first family of benchmarks were the following:

- **Memory-Saturation Threshold**: Determine the level of concurrency at which the KV-cache exhausts available GPU memory.
- **Throughput under Load**: Quantify how aggregate tokens-per-second changes with rising concurrency and assess the extent to which memory saturation constrains peak throughput.
- **Latency Scaling**: Measure the growth in latency the request count increases.
- **Queue-Formation Dynamics**: Observe when and how request queues build up in the scheduler and relate their depth to concurrency and cache pressure.
- **Impact of Model Size**: Compare these effects across models of different parameter counts to reveal how model scale shifts the saturation point and alters throughput and latency trends.

In order to conduct these experiments, the following configurations were utilized:

Parameter	Value
Number of Nodes	1
Number of GPUs (tensor parallelism)	1
User Prompt Length	128 tokens
Model Output Generation Length	512 tokens
KV-cache dtype	FP16
Metrics Collection Interval (Prometheus)	∼200 ms

Table 5.2: Benchmark setup and hyper-parameters for the concurrency sweep

These benchmarks were executed on a single node and tensor-parallelism set to one, guided by two considerations. First, the goal was to push each model to its concurrency limit, by finding the point at which hardware resources became exhausted. Because of this, restricting the testbed to a single GPU allowed for the resource limits to appear sooner, and required far fewer simulated user requests to reach. If multiple nodes or multiple GPUs had been utilized, the hardware ceiling would have been pushed much higher, forcing a much larger traffic load before the same bottlenecks emerged. Second, a single-GPU setup avoids the communication overhead that would arise in multi-GPU or multi-node configurations. Eliminating those extra latencies ensured that any trends in response time remained attributable to scheduler pressure and KV-cache usage, not to network contention or collective-communication delays.

Another chosen setting was the length of the user requests, measured in tokens. A prompt length of 128 tokens and maximum generated output of 512 tokens was selected, to reflect a typical interactive use case while keeping individual requests modest in size. Every request in the concurrency sweep used these same limits, ensuring that input and output length remained constant across all traffic levels. This uniformity allowed the experiment to isolate the effect of increasing concurrency on hardware utilisation and latency without confounding factors introduced by variable sequence lengths.

Furthermore, all models were served in FP16 format, a widely adopted industry standard that balances numerical precision with memory efficiency. Using the same dtype for every model prevented quantisation differences from influencing the latency and throughput measurements.

Lastly, preliminary tests were conducted to determine the best metric-scraping frequency. The concluded result was to scrape and save the Prometheus hardware metrics every 200ms, since this interval provided a satisfactory temporal

resolution to capture load fluctuations while imposing negligible overhead on the system.

Model	Concurrency Levels Tested (number of requests)	VRAM (GiB)	VRAM left for KV-Cache (tokens)	Deploy Time (s)
gpt2-xl	1, 10, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1 000, 1 100, 1 200	2.9676	268 208	1.813296
deepseek-7B-chat	1, 10, 50, 100, 200, 300, 400, 500, 600	12.8726	141 808	4.415860
Qwen-14B-Chat	1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160	26.4278	64 912	3.361213
Falcon-40B	1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160	76.9338	18 688	8.945311

Table 5.3: Concurrency-sweep configurations for each of the selected models

A further aim of the concurrency experiments was to examine how model size influences performance. Larger models store more parameters and, as a result, each token occupies more memory; fewer tokens can therefore reside in the KV-cache at any one time, limiting the number of concurrent requests the system can handle. Table 5.3 confirms this relationship. When each model was deployed, the available KV-cache decreased as model size increased. For example, GPT-2 XL, with 1.5 billion parameters, accommodated 268 208 tokens, whereas the 40-billion-parameter Falcon model could hold only 18 688 tokens. Because all users share this finite memory pool, a smaller cache directly reduces the maximum number of simultaneous requests the server can support. The implications of this constraint, and associated findings, are discussed in detail in the Results chapter.

Metric ID	Prometheus Type	Unit
gpu_cache_usage_perc	Gauge	% (no unit)
num_requests_waiting	Gauge	Requests
request_queue_time_seconds	Histogram	Seconds
time_to_first_token_seconds	Histogram	Seconds
time_to_first_token_seconds_sum	Counter	Seconds
time_to_first_token_seconds_count	Counter	Count
time_per_output_token_seconds	Histogram	Seconds
generation_tokens_total	Counter	Tokens
timestamp	Gauge	Seconds

Table 5.4: Most relevant Prometheus metrics to record for the concurrency benchmarks

For every experiment, more than 300 hardware metrics were collected every 200 ms. However, for the concurrency-related benchmarks, Table 5.4 lists the metrics that best support the stated objectives. For instance, *gpu_cache_usage_perc* indicated the concurrency level at which the KV-cache exhausted available GPU memory, while *time_to_first_token_seconds_sum* and *time_to_first_token_seconds_count* were used to compute the mean time-to-first-token latency for each level of concurrent requests. In addition, metrics such as *num_requests_waiting* and *request_queue _time_seconds* were used to track queue growth by reporting both the number of pending requests and their waiting times.

The analysis and interpretation of these concurrency-related benchmarks are detailed in Section 6.1, under the Results chapter.

5.1.2 Sequence Length-Related Benchmarks

In real-world large-language-model inference, user prompts may range from brief search queries to entire contracts or chat transcripts that expand with each turn. For example, a user might ask the model to translate a short phrase, whereas another might request the same model to process a full document and summarise it. Each of these tasks has completely different characteristics, yet the system must adapt to all.

The length of the model's response is likewise variable. Some users expect concise answers, while others request detailed technical explanations, step-by-step reasoning, or multi-paragraph summaries.

Because neither prompt length nor output length can be known in advance and both can vary widely from one request to the next, adjusting the underlying infrastructure to perform well across this spectrum is increasingly challenging.

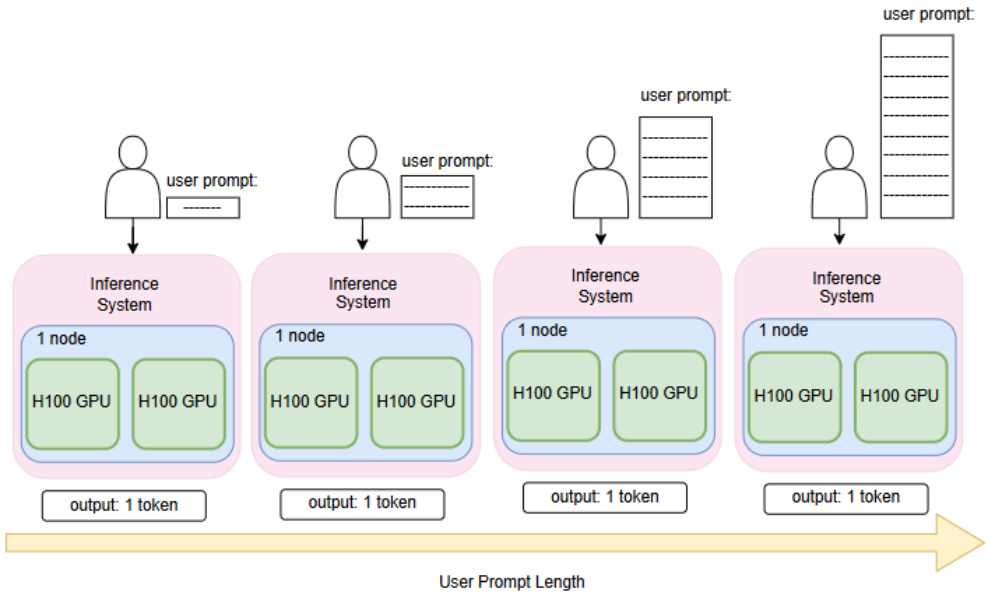


Figure 5.3: Sequence Length Setup: Increasing User Prompt Length Served to the Inference System

This has driven the motivation to study this second family of benchmarks, related to the sequence-length of user requests and model outputs. The objective of this benchmark family is therefore to determine how varying sequence length, in both the user prompt and the model's generated output, reshapes overall system behaviour. Specifically, the study pursued the following objectives:

- **Prompt-Length Impact**: Quantify how expanding the input sequence up to 128 k tokens affects time-to-first-token latency and prefill throughput.
- **Generation-Length Impact**: Measure how extending the maximum output length influences end-to-end latency and tokens-per-second decoding rate, thereby assessing decoder efficiency.
- **Streaming Benefit**: Compare streaming and non-streaming responses for a fixed workload in order to determine the reduction in user-perceived response time against the non-streaming baseline.

To address these objectives, the study was organised into the three experiments summarised in Table 5.5. In the first experiment the model's maximum generation

length was fixed to a single token, while the user-prompt length increased from 8 k to 128 k tokens across successive runs.This configuration was chosen because it allowed to isolate the impact of input size on prefill latency.

The second experiment focused on decoder behaviour: the prompt length was held constant at 128 tokens and the permitted output length was expanded step by step up to 128 k tokens, thereby exposing decode-time efficiency.

Finally, a third experiment compared user-perceived latency with and without streaming by running the same workload twice; once with the streaming flag enabled and once with it disabled.

Experiment	**User Prompt Length**	**Output Length**	**Streaming**
1	8 k, 16 k, 24 k, 32 k, 40 k, 48 k, 56 k, 64 k, 72 k, 80 k, 88 k, 96 k, 104 k, 112 k, 120 k, 128 k	1 token	True
2	128 tokens	8 k, 16 k, 24 k, 32 k, 40 k, 48 k, 56 k, 64 k, 72 k, 80 k, 88 k, 96 k, 104 k, 112 k, 120 k, 128 k	True
3	64 k tokens	64 k tokens	False / True

Table 5.5: Sequence-length benchmark matrix

The following configurations were utilized for these sequence length-related benchmarks:

Parameter	Value
Number of Nodes	1
Number of GPUs (tensor parallelism)	2
Model Used	Mistral-7B-128k
Context Window	128 k tokens
Number of Concurrent User Requests	1
Model VRAM Occupation	GPU0: 6.78 GiB GPU1: 6.78 GiB
VRAM left for KV-Cache (tokens)	GPU0: 1 194 848 tokens GPU1: 1 194 848 tokens
Model Deployment Time	5.315 s
KV-cache dtype	FP16
Metrics Collection Interval (Prometheus)	∼200 ms

Table 5.6: Benchmark setup and hyper-parameters for the sequence-length experiments

Out of all the configurations necessary for these benchmarks, a model with an exceptionally large context window was particularly crucial, since the experiments required handling very long prompts and equally extensive generations. Alternatives such as GPT-2-XL or DeepSeek-LLM-7B-Chat provide much smaller context limits, which would not have allowed a broad enough range of sequence lengths to yield meaningful conclusions. Mistral-7B-128k was selected for this reason: its 128-k-token window greatly exceeds that of the other models used in this study, as shown in Table 5.1.

Metric ID	Prometheus Type	Unit
vllm:gpu_cache_usage_perc	Gauge	% (no unit)
vllm:request_prompt_tokens_	Histogram	Tokens
vllm:request_generation_tokens_	Histogram	Tokens
vllm:time_to_first_token_seconds_	Histogram	Seconds
vllm:request_prefill_time_seconds_	Histogram	Seconds
vllm:time_per_output_token_seconds_	Histogram	Seconds
vllm:request_decode_time_seconds_	Histogram	Seconds
vllm:e2e_request_latency_seconds_	Histogram	Seconds
vllm:iteration_tokens_total_	Histogram	Tokens
vllm:prompt_tokens_total	Counter	Tokens
vllm:generation_tokens_total	Counter	Tokens
timestamp	Gauge	Seconds

Table 5.7: Most relevant Prometheus metrics for the sequence-length benchmarks

Lastly, Table 5.7 details the metrics of interest for this second family of benchmarks. Since latency can be divided into into prefill, decode, and end-to-end measurements, metrics such as *request-prefill-time-seconds*, *request-decode-time-seconds*, and *e2e-request-latency-seconds* were key in order to analyse each one independently.

The analysis and interpretation of these sequence length-related benchmarks are detailed in Section 6.2, under the Results chapter.

5.1.3 Intra-node Scaling-Related Benchmarks

Tensor-parallelism may be pictured as an assembly line in which two identical stations share the same task: instead of a single worker multiplying all of the matrices that drive a 70-billion-parameter model, each GPU processes one-half of the computation, handing partial results to its partner until a token is completed. In theory this duplication should halve the time required to produce each token; in practice, the extra coordination and data exchange can erode much of that ideal speed-up, so the true benefit must be measured rather than assumed.

The question matters because modern applications such as chat assistants, code generation tools and search augmentation impose latency budgets of only a few hundred milliseconds, meaning that even a modest improvement in aggregate

tokens-per-second can translate into a perceptibly smoother user experience when thousands of requests are served in parallel.

However, it is also important to take into account the fact that every additional GPU represents a concrete cost in power, cooling and rack space. Therefore, if tensor-parallelism of two does not deliver commensurate returns under typical traffic, operators may prefer to use a fleet of single-GPU nodes rather than fewer multi-GPU boxes for their data centers.

This has driven the motivation to study this third family of benchmarks, related to Intra-node Scaling. The objective of this benchmark family is to quantify the real-life trade-offs of single vs double tensor-parallelism in LLM inference systems. Specifically, the study pursued the following objectives:

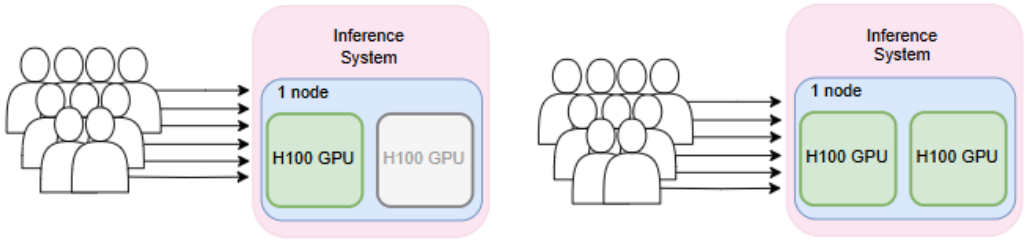


Figure 5.4: Intra-node Setup: Increasing Tensor Parallelism in the Inference System

- **Throughput Scaling**: Measure the change in aggregate tokens-per-second when tensor parallelism is increased from 1 to 2 and determine whether the additional GPU delivers proportional gains.
- **GPU-Utilisation Balance**: Examine utilisation metrics on both devices to assess how evenly compute and memory bandwidth are exercised under tensor parallelism equal to 2.
- **KV-Cache Distribution**: Observe how splitting the model affects available KV-cache per GPU and identify whether this alleviates or merely shifts memory pressure.
- **Concurrency Efficiency**: Verify that throughput improvements, if any, persist across several concurrency levels rather than only at the single traffic point used for comparison.
- **Cost–Benefit Analysis**: Combine the above findings to decide whether the marginal performance gained by adding a second GPU justifies its additional

power and occupancy within the node.

In order to conduct these experiments, the following configurations were utilized:

Parameter	**Value**
Number of Nodes	1
Number of GPUs (tensor parallelism)	1 GPU / 2 GPUs
Model Used	Meta-Llama-3.1-70B-Instruct-FP8
Context Window	128 k tokens
KV-cache dtype	FP8
Metrics Collection Interval (Prometheus)	∼200 ms
User Prompt Length	1 000 tokens
Model Output Generation Length	128 tokens

Table 5.8: Benchmark setup and hyper-parameters for the intra-node (tensor-parallel) scaling study

Parameter	**Value**
Tensor Parallelism	1
User Prompt Lengths*	1, 10, 20, . . . , 250
Model Output Generation Length	128 tokens
Model VRAM Occupation	67.70 GiB
VRAM left for KV-cache	29 648 tokens
Model Deployment Time	7.873756 s

Table 5.9: Run-specific configuration for the TP = 1 experiments

Parameter	Value
Tensor Parallelism	2
User Prompt Lengths*	1, 10, 20, . . . , 250
Model VRAM Occupation	GPU0: 33.87 GiB GPU1: 33.87 GiB
VRAM left for KV-cache	GPU0: 278 496 tokens GPU1: 278 496 tokens
Model Deployment Time	9.416891 s
Model Output Generation Length	128 tokens

Table 5.10: Run-specific configuration for the TP = 2 experiments

As it can be observed in Table 5.8, Meta-Llama-3.1-70B-Instruct was paired with FP8 weights because this precision keeps the 70-billion-parameter model small enough to fit on one GPU and, at the same time, lets the model be split cleanly across two GPUs when tensor parallelism is enabled. Every request carried a prompt of exactly 1 000 tokens, and the model was allowed to emit at most 128 tokens. Fixing these lengths removed sequence-size as a source of variation, so throughput changes could be traced only to the level of parallelism. The benchmark ran twice for each traffic level: first with tensor parallelism set to 1 and then with it set to 2. Aggregate tokens-per-second were collected in both cases. To check whether any speed-up remained stable as load grew, the same test was repeated automatically for 25 different concurrency points, from 1 up to 250 simultaneous requests. This setup held every parameter constant except the number of GPUs that shared the model, allowing a clear view of how intra-node tensor parallelism affected throughput, GPU memory balance and overall efficiency.

Metric ID	Prometheus Type	Unit
vllm:iteration_tokens_total_	Histogram	Tokens
vllm:prompt_tokens_total	Counter	Tokens
vllm:generation_tokens_total	Counter	Tokens
vllm:e2e_request_latency_seconds_	Histogram	Seconds
vllm:request_prefill_time_seconds_	Histogram	Seconds
vllm:request_decode_time_seconds_	Histogram	Seconds
vllm:gpu_cache_usage_perc	Gauge	% (no unit)
vllm:num_requests_running	Gauge	Requests
process_resident_memory_bytes	Gauge	Bytes
timestamp	Gauge	Seconds

Table 5.11: Key Prometheus metrics used for the intra-node scaling benchmarks (tensor parallelism 1 vs 2).

The set of metrics captured for this third benchmark family was limited to those that directly reflect throughput, latency, and memory balance, which are the core concerns of the intra-node study. For instance, *iteration-tokens-total* records every token produced within each sampling window; dividing that figure by the window duration yields aggregate tokens per second and makes it possible to track how throughput changes between TP 1 and TP 2. Likewise, pairing *gpu-cache-usage-perc* with *process-resident-memory-bytes* shows how the KV-cache is distributed across the two GPUs and whether the second device truly relieves pressure or merely mirrors it.

To conclude, the analysis and interpretation of these intra-node scaling-related benchmarks can be read in Section 6.3, under the Results chapter.

5.2 Pipeline Orchestration

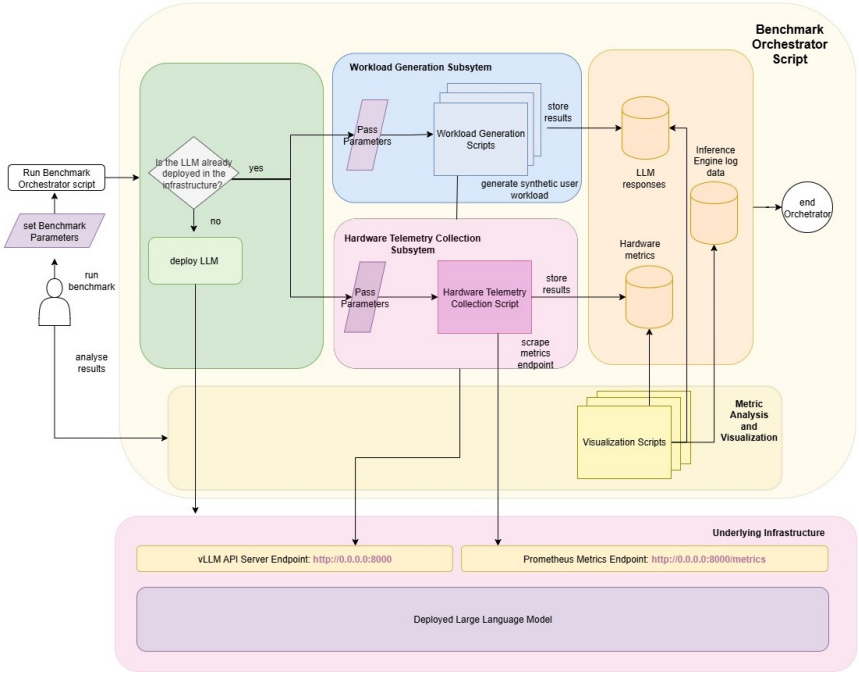


Figure 5.5: Pipeline Orchestration Schema

5.2.1 Purpose and Motivation

Running this project was expected to involve hundreds of experiments, each with its own configuration and operational parameters. Additionally, each experiment involves several subsystems working together, including model deployment, user workload generation, telemetry capture, and results aggregation. Since each run was anticipated to produce large and varied outputs, such as telemetry logs and metric dumps, it was clear that without a unifying approach, coordinating these experiments would become slow and inconsistent, as well as prone to error. Running everything manually would likely lead to misplaced files, inconsistent results, and timing issues between subsystems. Although each part could run on its own, without proper coordination they would not follow the same naming rules or start and finish in sync, making the results harder to manage.

To address these challenges, the decision was made to design a system that would act as an orchestration layer for the benchmarks. This layer was responsible for coordinating the execution of all subsystems, ensuring that they operated in the correct sequence and under consistent conditions. It also centralised the control

of configuration parameters, allowing new experiments to be defined and executed without modifying the underlying subsystem code.

The following requirements were defined for the orchestration system:

- Automate the execution of all subsystems to remove the need for manual coordination.
- Maintain modularity so each subsystem could be invoked and maintained independently.
- Produce clean and consistently structured output to simplify post-processing and analysis.
- Ensure reproducibility so that experiments with identical configurations follow the same execution path in the pipeline.
- Include fault tolerance mechanisms to handle and recover from subsystem failures gracefully.

5.2.2 High-Level End-to-End Flow

At a conceptual level, a single benchmarking run followed a clear and repeatable sequence of actions. This overview focuses on the logical flow rather than the implementation details, providing a top–down view of how the system operated from start to finish.

1. **Environment Preparation.** The orchestrator created the necessary directory structure for storing all results and logs, ensuring that each run could be traced back to its configuration.
2. **Model Availability Check.** It verified that the chosen large language model was active on the target infrastructure. If the model was not detected, the orchestrator initiated a complete deployment, applying all predefined parameters that influenced the underlying environment, including the selected open-source model and the tensor parallelism configuration chosen for that benchmark.
3. **Configuration Execution Loop.** For each experiment configuration (for example, a specific request count):
   (a) The telemetry scraper was started to capture metrics in real time.
   (b) The user workload generator was started in parallel, and was tasked with sending synthetic user requests to the deployed large language

model, following the specified configurations that detailed the parameters such as number of concurrent users or token limits for input prompts and model responses.

(c) Once the user workload completed, the telemetry scraper was stopped to finalise metric collection.

4. **Result Storage.** All outputs, which included raw telemetry, processed metrics, and workload logs, were saved using a consistent directory structure. This standardisation made post-processing and comparison across experiments straightforward.

   Observation: In practice, a single benchmark pipeline execution could include dozens of runs of the same benchmark, varying one parameter while keeping the others fixed. This allowed the effect of that parameter to be isolated and its influence on performance to be measured more accurately.

5. **Optional Model Shutdown.** After completing the scheduled benchmark, the orchestrator decided whether to terminate the inference engine based on its initial state. If the engine had been started by the orchestrator, it was shut down to free resources. If it had already been running before the benchmark began, it was left active.

Each stage of this flow was fully parameterised. Variables such as the model name, number of requests, output length, and output directory could be adjusted at the start of the orchestrator script, allowing the same system to adapt seamlessly to new benchmarking scenarios.

5.2.3 Architectural Decisions

Given the system requirements outlined in Section 5.2.1, the Pipeline Orchestration system was designed with the following architectural principles in mind:

- **Separation of Concerns:** The orchestration logic was kept separate from the telemetry and workload subsystems, ensuring that these components remained reusable and could be tested independently.
- **Parameter Passing:** Key parameters, such as `-model`, `-num_requests`, and `-output_length`, were supplied at runtime. This decoupled subsystem logic from benchmark-specific constants, allowing greater flexibility.
- **Parallel Execution:** The telemetry scraper ran concurrently with workload generation so that metrics reflected the same time window as the processed requests.

- **State Management:** A timestamp file was used to assign a common identifier to all results from a single run, simplifying traceability.
- **Idempotence:** Before starting a new instance, the system verified whether `vLLM` was already running, preventing unnecessary restarts.

5.2.4 Benefits of This Approach

The architectural decisions outlined above addressed the specific requirements of the orchestration system and had direct, measurable benefits:

- **Reproducibility:** Each benchmark followed the same automated process, which minimised human intervention and ensured that results could be compared under identical conditions.
- **Scalability:** New benchmarks, models, or hardware configurations could be added by changing parameters, without modifying the underlying code.
- **Traceability:** The directory structure preserved a clear mapping between configurations and their results, making retrieval straightforward during analysis.
- **Resource Awareness:** Storing telemetry output directly to disk prevented unnecessary GPU memory use, leaving more resources available for model inference. This links to the storage strategy discussed in Section **??**.

To summarize, this section outlined the overall design and operation of the Pipeline Orchestration system, providing a high-level view of its role, architecture, and benefits. The following subsections will examine the implementation of each component in detail, explaining how the design choices described here were realised in practice.

5.3 Workload Generation

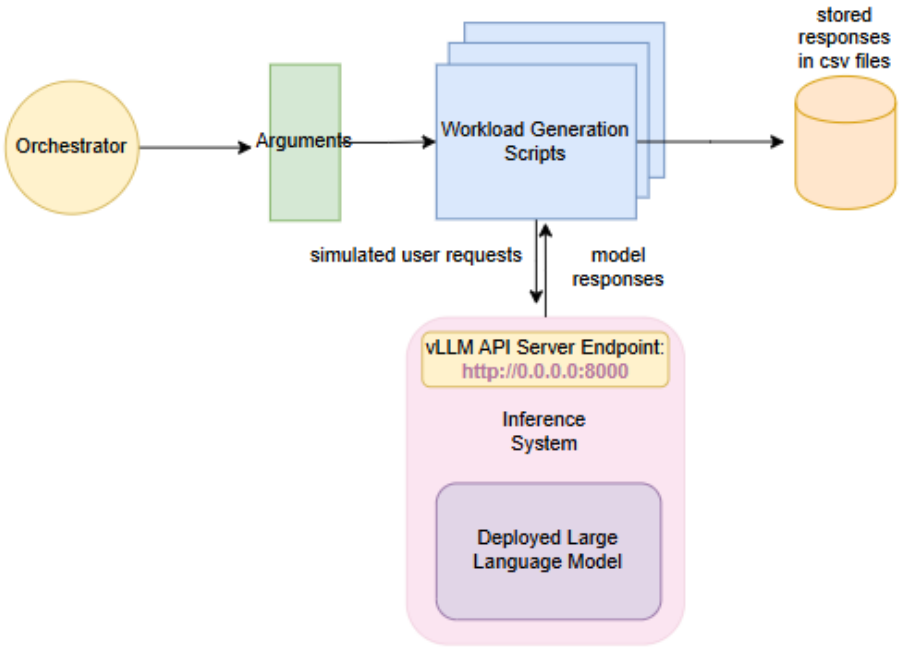


Figure 5.6: Workload Generation Sub-system Schema

5.3.1 Purpose and Role in the Benchmarking Pipeline

This section describes the workload generation subsystem, responsible for creating and sending synthetic user requests to the inference backend. These workloads emulated different usage patterns of users to stress the inference system under specific conditions such as high concurrency, growing sequence length, and varying tensor parallel configurations.

It is important to clarify that the goal of this subsystem was not to replicate the full variability of real-world user traffic, but rather to construct controlled workloads in which a single parameter of interest could be varied while others remain fixed. This methodological choice allowed for isolating the effect of specific factors, such as the number of concurrent requests, the length of the input prompt, or the configuration of tensor parallelism, and observing their direct impact on inference performance. In this sense, the purpose of the workload generation subsystem was not to mimic production heterogeneity, but to generate precise and repeatable user

input patterns that enable targeted performance investigations.

To serve this role effectively, the subsystem was designed to meet these core requirements:

- Allow workloads to be fully configurable, so that key parameters can be independently controlled and varied across experiments.
- Support automation, enabling the execution of multiple benchmarks in sequence without manual intervention.
- Ensure reproducibility, by making all inputs and configurations explicitly defined and reusable for consistent reruns.

The specific strategies used to fulfill these requirements, including how workloads are parameterized, automated, and logged, are described in the subsections that follow.

5.3.2 Model Access and Request Handling

Once the purpose of the workload generation subsystem has been established, it becomes essential to explain how requests were transmitted to the underlying inference engine. In the benchmarking system developed for this project, the model was hosted locally using a vLLM server, which exposed a REST API that replicated the structure and behavior of OpenAI's official API. Rather than manually crafting low-level HTTP requests, which would involve explicitly building payloads, setting headers, and parsing responses, the decision was made to integrate the official OpenAI Python client library as the interface responsible for issuing and handling inference requests.

```
from openai import OpenAI

# Configure the client to target the local vLLM server
client = OpenAI(
    api_key="dummy",    # vLLM does not validate this
    base_url="http://localhost:8000/v1"
)

# Issue a concise completion request
resp = client.completions.create(
    model="RedHatAI\Meta-Llama-3.1-70B-Instruct-FP8",
    prompt="Hello " * 128,     # ~128-token prompt
    max_tokens=64,
    temperature=0.0
```

```
)

text = resp.choices[0].text.strip()
```

Listing 5.1: Example of OpenAI client pointed to the local vLLM endpoint and used to issue a request.

The decision to use the OpenAI client in this project was deliberate and driven by both practical and architectural considerations. At its core, the client provided a clean, standardized way to communicate with any server that followed the OpenAI API specification. This meant that no custom networking logic needed to be written. The only configuration required was specifying a different base URL, which in this case pointed the client to the locally hosted vLLM server at `http://localhost:8000/v1`. Since this was defined in the configuration rather than in the code itself, the scripts worked for both local and remote deployments without needing any modifications.

Once connected, the client's `completions.create()` method was used as a high-level interface for sending inference requests. Here, essential parameters such as the prompt, model name, maximum token count, and temperature could be set in a concise, readable way. This avoided low-level request construction and made the workload generation scripts easier to read, maintain, and adapt.

Equally important was the client's built-in validation and error handling. These features automatically checked the integrity of responses and managed common connection issues, removing the need to manually implement these safeguards. From a software engineering perspective, this design choice aligned with best practices: keeping request logic modular, ensuring fault tolerance, and preserving the ability to switch between deployment environments with minimal effort.

Overall, the use of the OpenAI client served a practical and architectural role. It enabled standardized access to the inference server while abstracting away the mechanics of request construction and transmission. This design choice aligned with the broader goals of the benchmarking system, which emphasized clarity and reusability. The resulting implementation allowed requests to be issued efficiently and consistently, supporting a variety of workloads while remaining adaptable to changes in infrastructure or deployment strategy.

5.3.3 Fundamental Benchmarking Unit: Single Request Flow

Figure 5.7 illustrates the end-to-end flow of a single simulated user request launched from the workload generation subsystem. This represents the smallest executable unit within the benchmarking process, being one request from one user, and serves

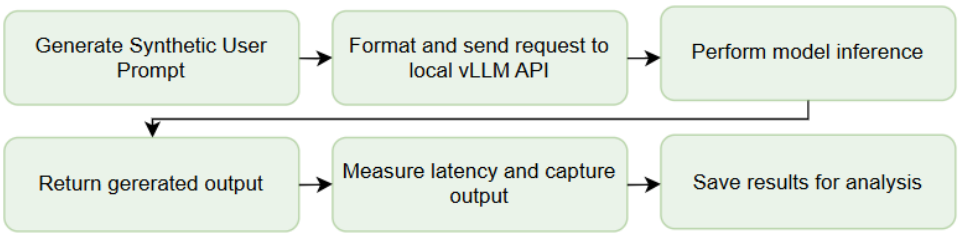


Figure 5.7: View of the end-to-end flow for a single simulated user request

as the foundation upon which larger experiments are built. In practice, each full benchmark consisted of hundreds of such requests executed in sequence or in parallel, with only one variable (for example, prompt length or number of concurrent users) changing between runs. Presenting the single-request flow first makes it possible to understand the fundamental mechanics before examining more complex scenarios involving large-scale parameter sweeps. At this level, the process is straightforward and fully controlled. A prompt is first defined, either manually or programmatically depending on the benchmark configuration. The prompt is then sent to the locally hosted vLLM server via the OpenAI client, which abstracts the details of request construction, transmission, and response parsing.

Once the request reaches the vLLM server, it is processed by the large language model which was previously selected and deployed into the infrastructure. Behind the scenes, this involves tokenizing the input prompt, running it through the model's transformer layers, and using the prebuilt key–value (KV) cache to efficiently handle attention computations for each generated token. The model produces an output sequence, which is returned to the workload generation subsystem. At this point, the total time taken for the request is measured and used to calculate the latency. The generated output, together with its associated latency, is then saved to a results file for later analysis.

These per-request results complement the hardware-level metrics collected in real time by the metrics collection subsystem (described in Section 5.4), providing both application-level and system-level perspectives on performance within the same benchmark run.

5.3.4 Parameterization for Flexibility and Reproducibility

To support repeatable and configurable experimentation, the workload generation scripts were designed to receive their configuration through command-line arguments rather than hardcoded values. This choice allowed the same script to be

reused across a wide variety of benchmark scenarios, with no need to modify its internal logic between runs. Parameters such as the number of concurrent requests, the maximum number of output tokens, the model to be tested, and the directory in which results were saved could all be specified externally when launching the script.

This approach introduced several important benefits:

- **Flexibility:** It enabled rapid changes in workload configuration without requiring edits to the code.
- **Reproducibility:** The exact conditions of a benchmark could be clearly defined, stored, and repeated at any time.
- **Modularity:** The logic of how a benchmark operated was kept separate from the specifics of a given test case.
- **Reduced likelihood of error:** Avoiding hardcoded parameters removed the need for manual edits before each run, lowering the risk of inconsistency and misconfiguration.

5.3.5 Benchmark Configuration Parameters

As recently mentioned, the configuration of each benchmark run was controlled through a small set of command-line arguments that defined the behavior of the workload generation script. These arguments were selected to provide direct control over the key characteristics of the benchmark, making it easy to vary important parameters without modifying the script itself. The table below summarizes each argument, its purpose, and an example of how it was used:

Argument	Purpose	Example
`-num_requests`	Number of concurrent requests to simulate	`-num_requests 50`
`-prompt_length`	Number of tokens in the generated prompt	`-prompt_length 8192`
`-output_length`	Max tokens to generate per request	`-output_length 128`
`-model`	Name of the model to query	`-model gpt2-xl`
`-output_dir`	Directory where results will be saved	`--output_dir./results/{benchmark_family_name}/{timestamp}/{LLM_name}/{specific_config}`

Each of these arguments controls a different aspect of the workload. For example, increasing the number of requests turns a single-user scenario into a concurrency stress test. Changing the output length affects the amount of work the model must do for each input, while selecting a different model allows for comparisons across architectures or sizes. Specifying a unique output directory for each run ensures that results are not overwritten and remain easy to organize and trace. A it can be observed, other parameters that had to do with the underlying deployment strategy, such as –tensor-parallelism, were not needed for this script, bacause this subsystem was only in charge of generating the synthetic user requests that were sent to the infrastructure once i was already deployed. These other parameters, were used in other stages of the automated benchmarking pipeline in order to conduct the deployement of the underlying model.

```
import argparse

def parse_args() -> argparse.Namespace:
    """Parse benchmark parameters provided by the orchestrator at runtime."""
    p = argparse.ArgumentParser(
        description="Workload generation subsystem configuration."
    )
    p.add_argument("--num_requests",  type=int, required=True,
                   help="Number of concurrent requests to simulate.")
    p.add_argument("--prompt_length", type=int, required=True,
                   help="Number of tokens in the generated prompt.")
    p.add_argument("--output_length", type=int, default=128,
                   help="Max tokens to generate per request (default: 128).")
    p.add_argument("--model",         type=str,
    default="gpt2-xl",
                   help="Model to query during inference (default: gpt2-xl).")
    p.add_argument("--output_dir",    type=str, required=True,
                   help="Directory where results will be saved.")
    return p.parse_args()
```

Listing 5.2: Parsing benchmark configuration parameters inside the workload generation subsystem.

Within the benchmark orchestrator, the workload generation subsystem could be invoked directly as a standalone process by executing the command shown in Listing 5.3. All parameters required for the run were defined at the top of the orchestrator script, allowing the call to remain clean and free of hardcoded values. This structure meant that the same subsystem invocation could be reused multiple times within a loop inside the orchestrator, for example to incrementally vary the prompt length or the number of concurrent requests across successive runs. As a result, the configuration for an entire sequence of benchmarks could be adjusted in a single location, without modifying the subsystem's internal code or repeatedly editing the command itself. This design choice reflected one of the broader architectural goals of the benchmarking framework, which was to reduce manual intervention and allow parameter sweeps to be automated in a controlled and repeatable manner.

```
#!/bin/bash
# Orchestrator snippet: launches the workload generation
   subsystem with parameters.

python3 workload_generation_subsystem.py \
    --num_requests $NUM_REQUESTS \
    --prompt_length $PROMPT_LENGTH \
    --output_length $OUTPUT_LENGTH \
    --model $MODEL_NAME \
    --output_dir $OUTPUT_DIR
```

Listing 5.3: Example of the orchestrator invoking the workload generation subsystem with parameters defined as variables.

To summarize, this parameterization was meant to make the subsystem both expressive and user-friendly, since with only a few well-defined inputs, a wide variety of benchmark configurations can be created and executed in a consistent and controlled way.

5.3.6 Structured Output and Result Logging

A key design choice in the workload generation subsystem was the use of a user-defined output directory for storing benchmark results. By requiring the path to be specified through the `-output-dir` argument, the script avoided overwriting data from previous runs and enforced a clean separation between experiments. Each set of responses and measurements was saved in its own dedicated folder, making it straightforward to trace results back to the specific configuration that produced them.

The results were written in JSON Lines (JSONL) format, which was well-suited for storing structured data in a way that was both machine-readable and line-by-line parseable. This format facilitated efficient post-processing, especially in cases where results from many requests needed to be filtered, aggregated, or visualized. Keeping outputs in their own directories also simplified comparisons across runs, since each folder could be inspected or analyzed independently without ambiguity.

This structure proved especially valuable when benchmarks were launched in sequence through automation scripts. By assigning a unique output directory to each configuration, results could be collected systematically without requiring manual intervention. This organization strategy contributed directly to the reproducibility and scalability of the overall benchmarking pipeline.

5.4 Hardware Telemetry Collection

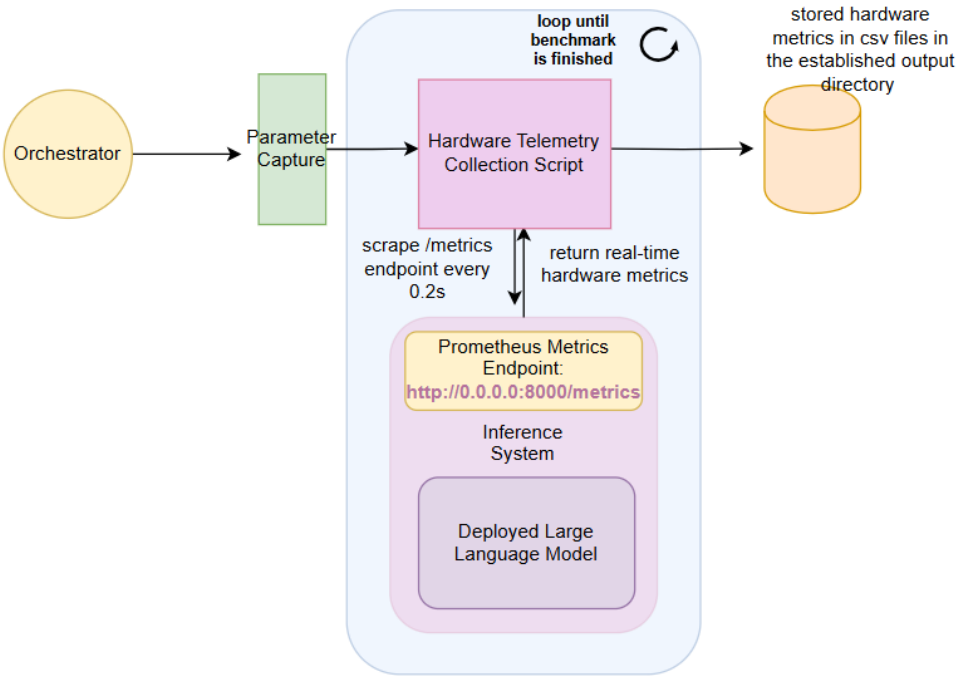


Figure 5.8: Hardware Telemetry Collection Sub-system Schema

5.4.1 Purpose and Role in the Benchmarking Pipeline

The hardware telemetry collection subsystem had the purpose of capturing hardware-level activity in real time while benchmarks were running. The need for collecting these low-level metrics arose from the fact that this project aimed to identify the underlying causes of inference performance, which could not be fully understood without examining detailed traces of metrics such as GPU utilization, memory allocation, and KV cache usage throughout the execution of each benchmark. Having this information available made it possible to establish direct correlations between changes in workload parameters and their immediate impact on the hardware, which in turn enabled a deeper understanding of performance characteristics and the identification of potential bottlenecks.

To serve this role effectively, the subsystem was designed to meet these core requirements:

- Capture hardware metrics in real time, at high temporal resolution during benchmark execution.
- Align data collection with workload execution windows, enabling direct correlation between system-level and application-level metrics.
- Minimize monitoring overhead, so that metric collection did not introduce measurable impact on inference performance.
- Store telemetry outputs in the same run-specific directory as the user workload results, allowing each benchmark's data to be traced back to its exact configuration.

5.4.2 Data Source and Collection Method

The hardware telemetry collection subsystem relied on the Prometheus metrics endpoint exposed by vLLM at `localhost:8000/metrics`. This endpoint, built into the vLLM serving framework, was chosen because it continuously reports detailed hardware and model statistics through a standardized interface. Using this built-in capability removed the need to modify the serving code and kept monitoring completely non-intrusive.

A main reason for using this endpoint was that it provided both GPU-level and model-level statistics in a single source. This made it straightforward to align hardware activity with the exact moment a request was processed. Other tools, such as `nvidia-smi` polling or NVIDIA Nsight profiling, could monitor GPU resources but would either require elevated permissions or introduce higher monitoring overhead.

They also lacked access to model-specific metrics like KV cache usage, which was critical for this study.

The metrics were presented in the Prometheus exposition format, which is plaintext and easy to parse programmatically. Each metric included descriptive names and labels, such as GPU device indices, which were necessary for correct interpretation in multi-GPU setups. To retrieve these metrics, the subsystem used the Python `requests` library instead of a full Prometheus client SDK. This kept the implementation lightweight and portable, allowing it to run in any environment without extra dependencies.

Collection was performed at a fixed rate of 0.2 seconds, resulting in five scrapes per second. This interval was chosen as a trade-off between temporal resolution and overhead. It was short enough to capture spikes in GPU utilization while keeping the impact on benchmark performance minimal.

```
timestamp,process_start_time_seconds,process_cpu_seconds_total,
vllm:num_requests_running,
vllm:num_requests_waiting,
vllm:gpu_cache_usage_perc{device="0"},
vllm:gpu_cache_usage_perc{device="1"},
vllm:gpu_prefix_cache_queries_total,
vllm:gpu_prefix_cache_queries_created,
vllm:gpu_prefix_cache_hits_total,
vllm:gpu_prefix_cache_hits_created,
vllm:num_preemptions_total,
vllm:num_preemptions_created,
vllm:prompt_tokens_total,vllm:prompt_tokens_created,
vllm:generation_tokens_total,vllm:generation_tokens_created,
vllm:request_success_total,vllm:request_success_created,
vllm:iteration_tokens_total,request_params_max_tokens,
vllm:time_to_first_token_seconds,
vllm:time_per_output_token_seconds,
vllm:e2e_request_latency_seconds,
vllm:request_queue_time_seconds,
vllm:request_prefill_time_seconds,
vllm:request_decode_time_seconds,
vllm:request_inference_time_seconds,
vllm:cache_config_info,http_requests_total
...
#For each benchmark run, the telemetry subsystem records more
   than 300 unique metrics at 0.2-second intervals. This
   listing only shows a representative subset.
0.0,  1.754207399e+09, 0.00,  0,  0,   0.0,  0.0,      0,
```

```
   0,    0,    0,    0,   0,     0,    0,      0,    0,    0,
    0,      0, 512, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000,
   0.000, 0.000, 0, 0
0.2,  1.754207399e+09, 0.15, 50,  5,  15.3, 14.7,    100,
   100,   95,   95,    0,   0,  6400, 6400,      0,    0,
   50,  50,   6400, 512, 0.062, 0.002, 0.064, 0.002, 0.040,
   0.022, 0.062, 0.062, 0,  50
0.4,  1.754207399e+09, 0.31, 50,  5,  20.4, 19.9,    200,
   100,  185,   90,    0,   0, 12800, 6400,      0,    0,
   100,  50,  12800, 512, 0.063, 0.002, 0.064, 0.002, 0.040,
   0.022, 0.062, 0.062, 0, 100
0.6,  1.754207399e+09, 0.48, 50,  5,  25.6, 25.1,    300,
   100,  270,   85,    0,   0, 19200, 6400,      0,    0,
   150,  50,  19200, 512, 0.064, 0.002, 0.065, 0.002, 0.041,
   0.022, 0.062, 0.062, 0, 150
0.8,  1.754207399e+09, 0.64, 50,  5,  30.7, 30.2,    400,
   100,  360,   90,    0,   0, 25600, 6400,      0,    0,
   200,  50,  25600, 512, 0.065, 0.002, 0.066, 0.002, 0.041,
   0.023, 0.063, 0.063, 0, 200
1.0,  1.754207399e+09, 0.80, 50,  5,  35.8, 35.4,    500,
   100,  450,   90,    0,   0, 32000, 6400,      0,    0,
   250,  50,  32000, 512, 0.065, 0.002, 0.066, 0.002, 0.041,
   0.023, 0.063, 0.063, 0, 250
1.2,  1.754207399e+09, 0.96, 50,  5,  40.3, 39.8,    600,
   100,  545,   95,    0,   0, 38400, 6400,      0,    0,
   300,  50,  38400, 512, 0.066, 0.002, 0.067, 0.002, 0.041,
   0.023, 0.063, 0.063, 0, 300

                              ...

4.8,  1.754207399e+09, 3.86, 50,  5,  99.8, 99.2,   2400,
   100, 2190,   90,    0,   0,122880, 6400,  51200, 6400,
   1200,  50, 128000, 512, 0.075, 0.003, 0.076, 0.003, 0.046,
   0.027, 0.070, 0.070, 0, 1200
5.0,  1.754207399e+09, 4.02, 50,  5, 100.0, 99.8,   2500,
   100, 2280,   90,    0,   0,128000, 6400,  56320, 6400,
   1250,  50, 134400, 512, 0.076, 0.003, 0.077, 0.003, 0.047,
   0.027, 0.070, 0.070, 0, 1250

                              ...
```

Listing 5.4: Example slice of the `metrics.csv` generated for one benchmark run; actual files include hundreds of metrics and span the entire run duration.

5.4.3 Metric Parsing and Storage

Once the raw metrics were retrieved from the Prometheus endpoint, they needed to be parsed and stored in a format that would be both complete and easy to work with during analysis. Each metric line from the endpoint included a name, an optional set of labels enclosed in braces, and its value. The parsing logic was designed to keep the metric name and its label set together as a single unique identifier, since separating them could lead to ambiguity when different devices or instances reported the same metric name. To achieve this, a regular expression was used to capture the entire metric name together with its labels before assigning the corresponding numeric value. This ensured that the column naming in the stored dataset remained unambiguous and traceable to the exact metric source.

The final output format for each benchmark run was a CSV file in which the first column recorded the elapsed time in seconds since the start of the run, and each subsequent column corresponded to one of the collected metrics. This choice was made because CSV is both human-readable and well supported by common data analysis tools such as Pandas for Python, which was later used to filter, aggregate, and visualize results. By storing the data in this way, the subsystem ensured that post-processing could be done efficiently without additional conversion steps, and most importantly, that the relationship between workload behavior and hardware state could be easily explored during analysis.

```
import re

# Regex to capture full metric name (with labels) and its value
_metric_re = re.compile(
    r'^([a-zA-Z_:][a-zA-Z0-9_:]*)'          # Metric name
    r'(\{[^}]*\})?'                          # Optional label set
    r'\s+([-+]?\d*\.?\d+(?:[eE][-+]?\d+)?)$' # Metric value
)

def parse_metrics(text):
    """Return dict: 'metric_name{labels}' -> float(value)"""
    metrics = {}
    for line in text.splitlines():
        if not line or line.startswith("#"):
            continue
        m = _metric_re.match(line)
        if m:
            name, labels, val = m.groups()
            metrics[name + (labels or "")] = float(val)
```

```
    return metrics

# Build complete header from all keys observed during run
all_keys = []
for _, m in records:
    for key in m:
        if key not in all_keys:
            all_keys.append(key)

writer.writerow(["timestamp"] + all_keys)
```

Listing 5.5: Parsing Prometheus metrics and dynamically constructing a union header to capture intermittent metrics.

During the implementation of the telemetry subsystem, an issue appeared when parsing certain metrics from the Prometheus endpoint. Some of these metrics were only exposed intermittently, depending on the execution stage or the GPU's internal state. If the CSV header was defined in advance, any metric that did not appear in the first scrape would be omitted entirely from the output. This meant that valuable data could be lost, and runs could not be compared reliably because each CSV file might have a different column structure.

To prevent this, the decision was made to adopt a dynamic "union header" approach. Instead of fixing the CSV header before collection, the header was built from the union of all metric keys encountered during the run, which ensured that even metrics appearing only once were also included in the final dataset. Additionally, this approach kept the structure consistent across benchmarks.

Lastly, all collected metrics were stored locally on the cluster's file system rather than being sent to a cloud service. This was an intentional architectural decision based on the nature of the benchmarking environment. The compute nodes operated within a controlled network segment where outbound internet access was not always guaranteed. In such settings, relying on a remote storage service would introduce uncertainty and additional points of failure. Keeping storage local removed any dependency on external connectivity and aligned with common High Performance Computing (HPC) practices, where network-isolated workloads are preferred for both security and reproducibility.

A potential concern with saving metrics locally is whether or not it would take up memory needed for later benchmarks. In practice, this was not an issue because the metrics were written straight to the node's disk, not stored in GPU memory. This meant that all GPU VRAM remained available for inference, with no loss of capacity or performance. As a result, collecting and storing telemetry had no

effect on throughput or latency in any benchmark run.

5.4.4 Output Organization

The hardware telemetry collected during a benchmark run was stored in the same run-specific output directory as the corresponding workload results. This design ensured that both application-level outputs, such as latency and throughput measurements, and hardware-level traces, such as GPU utilization and memory usage, were co-located in a single, self-contained folder. As a result, there was no need to cross-reference separate storage locations or rely on external mapping files to match telemetry data with its corresponding workload execution.

Placing the telemetry alongside the workload results also guaranteed a strict one-to-one correspondence between the two datasets. Every set of hardware metrics was inherently tied to a specific benchmark configuration and execution window, eliminating the risk of mismatched or misaligned data during analysis. This approach was especially important when running multiple benchmarks in sequence, as it prevented any accidental overwriting or mixing of results from different experiments.

This output structure had further benefits for reproducibility and post-run analysis. Since each run's data was fully encapsulated in its own directory, any benchmark could be revisited, re-analyzed, or shared without requiring additional context. The stored metrics could be examined in isolation or compared directly with other runs simply by inspecting their respective folders, making the organization both intuitive and scalable for large benchmarking campaigns.

5.4.5 Architectural Choices and Implications

This subsystem was built with a few clear architectural principles in mind. First, it was designed to work regardless of whether the benchmark was single-node, multi-node, or even running on a different serving backend. As long as a Prometheus endpoint was available, the scraper could capture metrics in real time. This effectively decoupled the telemetry subsystem from the model serving logic, making it reusable across a wide range of setups without modification.

Another key design choice was to avoid hardcoding output locations inside the telemetry subsystem. Instead, all storage paths were passed in as parameters from the benchmark orchestrator at runtime. This meant that the orchestrator could decide where results should be stored, keeping the telemetry logic clean and focused solely on data collection. It also ensured that both the telemetry subsystem and the workload generation subsystem could store their outputs in the same run-specific

directory. This made it easy to keep workload data and hardware telemetry side by side, which simplified later correlation and analysis.

By following these patterns, the subsystem remained portable, easy to integrate, and consistent in its behavior across different benchmarks. It could be triggered automatically by the orchestrator for any configuration, while still remaining a standalone tool that could be reused in other benchmarking workflows.

5.5 Metric Analysis and Visualization Subsytem

5.5.1 Purpose and Role in the Benchmarking Pipeline

The purpose of this last subsystem was to convert the raw, real-time data collected during the execution of the experiments into a format that was intuitive to interpret and from which conclusions could be drawn. While the specific metrics plotted varied according to the objectives of each benchmark, the underlying analysis and visualization framework remained unchanged.

This subsystem was designed to be flexible in how it integrated with the benchmarking pipeline. It could operate in two distinct modes:

1. **Automated mode.** If the orchestrator provided the path to the output directory where all results had been stored, the subsystem could retrieve the corresponding data files, process them, apply filtering, and generate the designated graphs automatically, storing them back in the same directory.
2. **Decoupled mode.** The subsystem could be run independently of the benchmarking pipeline to manually create new graphs or explore various metrics in a more case-specific manner, without being constrained by the automation layer.

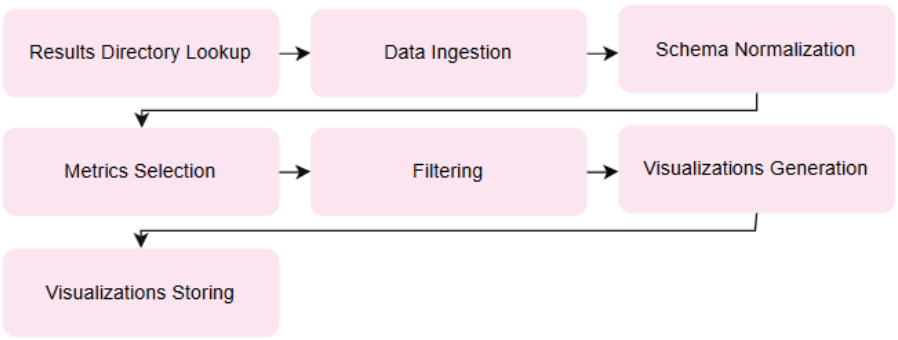


Figure 5.9: View of the end-to-end flow for the Metrics Analysis and Visualization subsystem.

5.5.2 Input Data Preparation

As already detailed in Section 5.4, every experiment in the benchmarking pipeline ultimately produced a standardized CSV log containing the complete set of metrics scraped during execution, regardless of the model, hardware configuration, or workload parameters involved. This was the result of a deliberate design choice to ensure that downstream analysis could be performed without having to account for differences in data format or metric availability between runs. The first responsibility of this subsystem was therefore to locate the correct results directory for the experiment and load the corresponding CSV into memory, ensuring that no relevant data source was overlooked. From there, the dataset was cleaned and normalized into a state suitable for analysis, which involved verifying consistent timestamp formats across all entries, preserving metric labels so that filtering could later be performed with precision, or simplifying column names when this improved readability without losing information. This preparation step was essential, as it transformed the raw output of the execution stage into a structured and coherent dataset that could serve as the foundation for every subsequent decision in the analysis process.

5.5.3 Metric Selection and Filtering

With the dataset prepared, the next task was to determine which signals would actually be examined, a decision that was guided by the specific objectives of the benchmark at hand. To maintain flexibility and reusability, metrics were identified dynamically through pattern matching in their names (for instance, locating any metric containing `time_to_first_token_seconds` or `gpu_cache_usage_perc`) rather than by relying on fixed column names tied to a specific run. This meant

that the same analysis code could be applied across very different scenarios without modification, even when the underlying models or workloads changed.

Once identified, these metrics were filtered to isolate only the relevant ones, which varied depending on the questions the benchmark was intended to answer, as detailed in Subsection 5.1. For example, in the case of latency-oriented studies, the metrics that resulted most important were those focused on TTFT, time per output token, and queue times. On the other hand, throughput studies prioritized token generation rates and concurrent request counts, and resource usage investigations emphasized GPU memory, KV-cache utilization, and other similar indicators.

5.5.4 Visualization and Output Storage

The final stage in this subsystem was concerned with translating the curated metrics into visual outputs, such a plots, histograms, dual-axis overlays, and others. The objective was that the metrics could be interpreted at a glance without the reader needing to read through large numeric tables. While the choice of specific plots was dictated by the focus of the benchmark, the visualization patterns themselves were intentionally kept consistent across experiments to support comparability.

Once generated, all visualizations were stored back into the same results directory alongside the processed datasets, keeping each run's artifacts self-contained and easy to revisit. This completed a design loop in which the data flowed from raw collection, through preparation and selection, into a form that directly answered the benchmark's original questions.

5.5.5 Abstraction Benefits

The analysis layer was intentionally designed to operate independently from the telemetry collection subsystem. This separation was a choice, and it allowed the analysis logic to remain agnostic to the origin, structure, or context of the metrics it processed. By avoiding any hard link between the two stages, it became possible to reuse the same analysis framework across very different experimental conditions without much modification.

This decision resulted in the following benefits:

- **Broad applicability:** The system could work with metrics from any type of benchmark, regardless of whether it focused on latency, throughput, or resource utilization, since the underlying analysis logic did not assume a fixed set of inputs.

- **Backend flexibility:** It could accommodate different serving backends, provided they exposed Prometheus-compatible endpoints, making it possible to integrate new models or infrastructures without changes to the analysis code.
- **Instant adaptability:** New metrics could be incorporated immediately into the analysis simply by matching their names, removing the need to rewrite parsing routines whenever an experiment introduced new telemetry signals.

In summary, the Metrics Analysis and Visualization subsystem converted the collected telemetry into readable plots and statistics, with an approach that stayed consistent across benchmarks while allowing straightforward adaptations to adjust to different experimental setups.

Chapter 6

Analysis and Interpretation of Results

In the following chapter, the results produced by the benchmark suite, executed after the system described in Chapter 5 had been implemented, are examined in detail and their performance implications are discussed.

Just like presented in Section 4.1, the accelerating volume of generative-AI traffic is placing unprecedented pressure on existing data-centre infrastructures. Because meeting that demand requires clusters of state-of-the-art GPUs whose acquisition and operation entail multibillion-dollar spendings, maximising the utilisation of memory and compute resources has become imperative. This project tackles that challenge by running a comprehensive benchmark suite that stresses LLM inference and measures how memory pressure, workload size, and cluster scale together constrain throughput and latency.

6.1 Analysis of Concurrency-Driven Performance Scaling

6.1.1 Analysis of GPU-Memory Saturation Across Concurrency Levels

In this first benchmark, the analysis focused on how increasing the number of concurrent user requests affected GPU memory utilization, thereby laying the groundwork for the later experiments that linked memory pressure to the throughput and latency experienced by users, which is largely driven by KV-cache contention and

scheduling overhead.

The goal was to perform a controlled stress test of the GPU memory and determine how many simultaneous requests it could sustain before saturation occurred. In order to achieve that, the benchmark gradually raised the concurrency level while recording the occupancy of the KV cache and the overall device-memory footprint at each step.

The initial model deployed was GPT-2-XL, which contained 1.5 billion parameters. It was hosted on a single node equipped with one H100 GPU offering 93.6 GiB of available memory. A single-node setup was chosen deliberately to eliminate inter-node latency and other distributed factors that might have distorted the measurements. By limiting the deployment to a single accelerator, the total memory pool was also constrained, which in turn reduced the number of concurrent requests required to reach saturation.

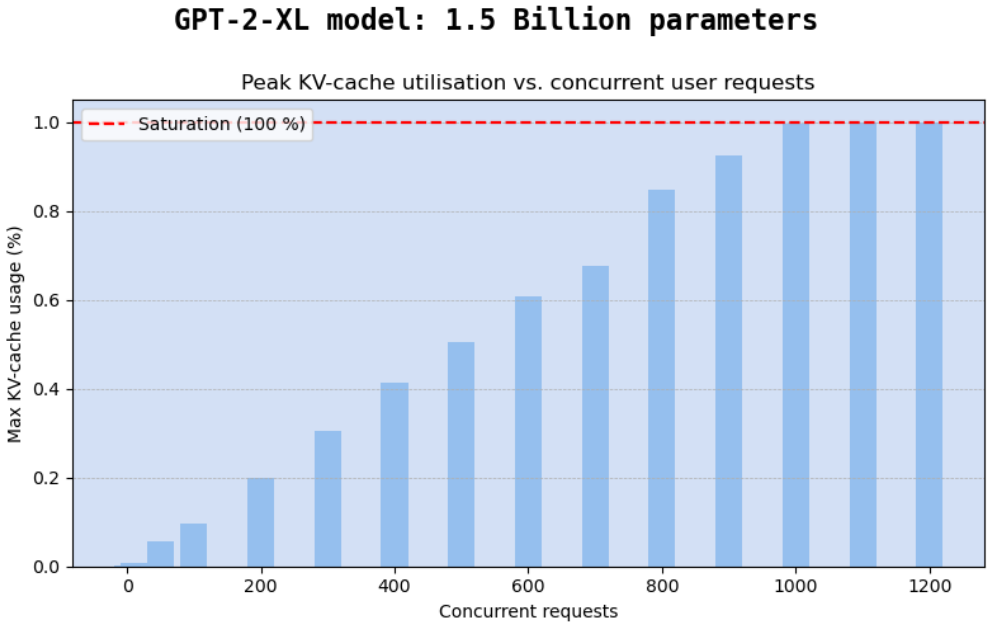


Figure 6.1: GPU KV-cache saturation curve for GPT-2-XL with increasing user requests

As shown in Figure 6.1, GPU-memory utilisation rose almost linearly as the number of concurrent requests increased from 1 to 1 000. At about 400 simultaneous requests the footprint reached roughly 40% of the H100's 93.6 GiB capacity. When concurrency hit 600, utilisation climbed to 60%, corresponding to 56.16 GiB of occupied memory. The trend continued until the cache neared saturation around

1,000 requests.

This proportional growth can be attributed to the fact that each prompt contained approximately 128 tokens and the model's maximum output length was capped at 512 tokens. Because both lengths remained constant, every request occupied an identical slice of the key–value cache. Increasing concurrency therefore meant that the GPU accumulated additional slices of equal size, which pushed the memory counter upward in a nearly straight line. The observation confirmed the expected behaviour that, with a steady token budget, KV-cache consumption scales linearly with the number of active requests.

Another observation drawn from Figure 6.1 was that memory utilisation grew linearly only until full saturation was reached, that is, until 100% of the available memory was consumed. For this 1.5-billion-parameter model running on a single H100 GPU, the safe operating range lay between 1 and roughly 1,000 concurrent requests. Beyond that level, additional requests saturated the memory. Furthermore, if users at a given concurrency increased prompt length, requested longer responses, or expanded the context window, the cache would have reached 100% utilisation even sooner.

These results revealed that the number of user requests processed concurrently had a direct impact on GPU-memory utilisation. Because GPU memory is finite, identifying the saturation point is essential for designing a batching policy that stays within the safe operating zone.

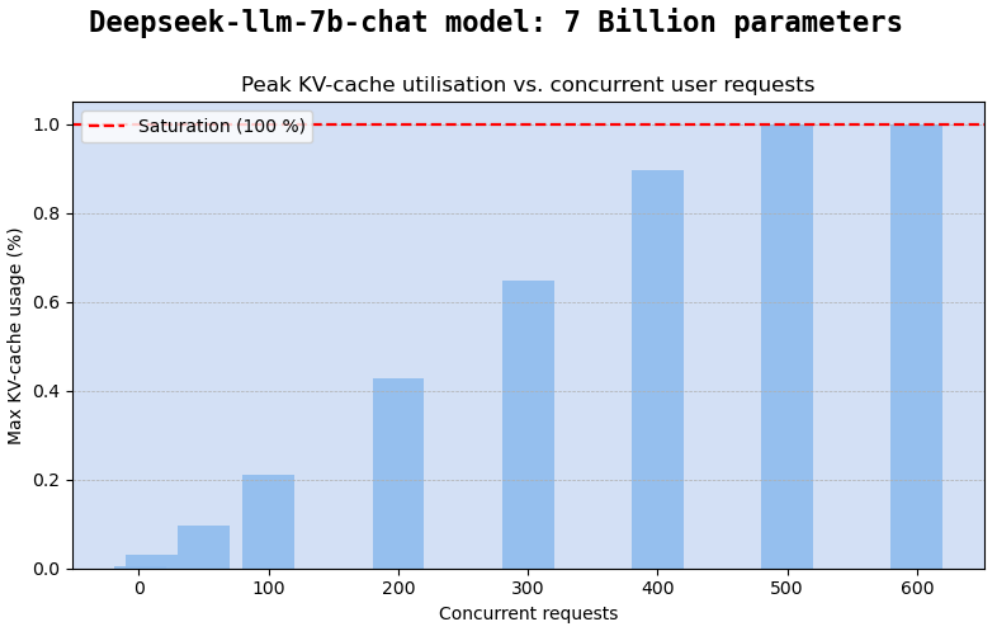


Figure 6.2: GPU KV-cache saturation curve for DeepSeek-LLM-7B with increasing user requests

This benchmark also meant to analyse how the choice of model and its parameter scale, including both the count and dimensionality, affected the number of concurrent requests the system could handle. To illustrate this effect, a second open-source model, DeepSeek-LLM-7B-Chat, was deployed with the same hardware configuration described earlier. DeepSeek contained about 7 billion parameters, whereas GPT-2-XL carried only 1.5 billion. Larger models typically incorporate more layers and wider embedding dimensions to improve reasoning capability, but this expanded architecture also increases the memory required to store a single token. The relationship among these factors can be derived from the formula presented and discussed in Section 3.5.1.

Total size of KV Cache (in bytes) :

$$2\, b\, t\, n_{\text{layers}}\, n_{\text{heads}}\, d_{\text{head}}\, P_a$$

The number of transformer layers in the model, n_{layers}, as well as the attention heads per layer, n_{heads}, and the vector dimension of each of those heads, d_{head}, are all dependent on the particular model that is deployed.

In Figure 6.2, it was proved how Deepsek-LLM-7B-chat, with a larger count for all transformer-related specifications, achieved a larger use of KV cache memory

usage than GPT2-XL, for the same number of concurrent requests. For example, for a number of 200 concurrent user requests, the GPT2-XL had to occupy 20% of the total memory, while Deepsek-LLM-7B-chat's memory usage mounted to more than 40% of the total available GPU memory.

This benchmark also revealed the precise load at which the DeepSeek-LLM-7B-chat model exhausts the available KV-cache: saturation was observed once concurrency reached roughly 500 simultaneous requests, with the cache already at 90% utilisation near 400. For this 7 billion parameter model, memory rather than compute becomes the dominant bottleneck once concurrency crosses the 500-request threshold. If user prompt length and model result length had to remain untouched, other practical options of reducing KV witching the KV cache to lower-precision formats such as FP8 (8-bit floating point) or BF16 (16-bit bfloat), which cut the cache footprint by 4× and 2× respectively, or distributing the model over additional GPUs so the cache can be sharded. These momory optimiation techniques are discussed in detail in Section 3.5.2.

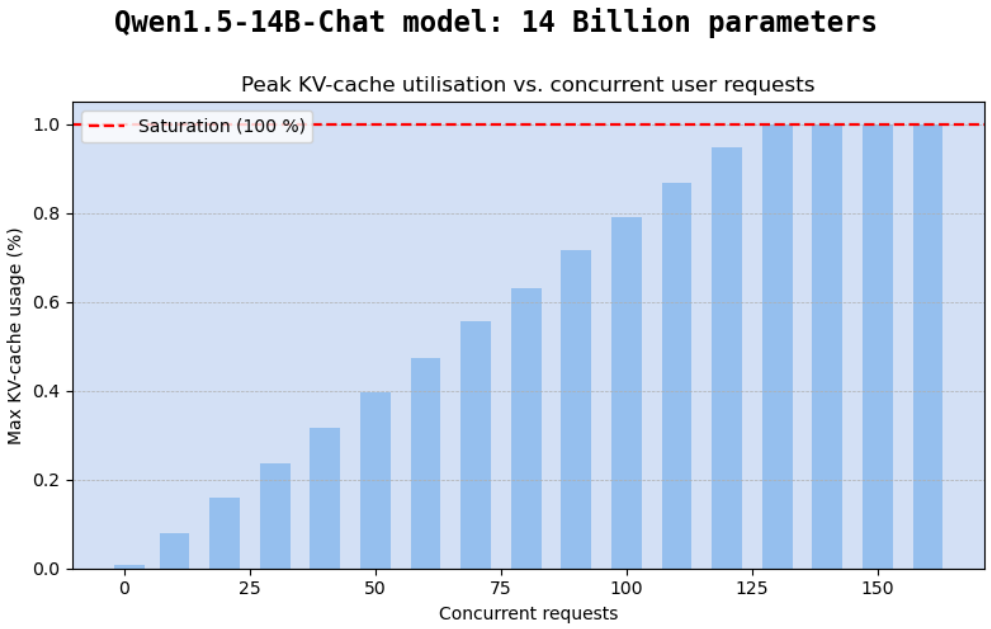


Figure 6.3: GPU KV-cache saturation curve for Qwen1.5-14B-Chat with increasing user requests

A third model, Qwen-1.5-14B-Chat, with approximately 14 billion parameters, was also deployed so that the earlier findings could be validated on a substantially larger architecture and the conclusions strengthened. Examining a network roughly twice the size of DeepSeek-LLM-7B and four times larger than GPT-2-XL

made it possible to verify whether the link between model capacity and KV-cache pressure remained consistent.

The curve in Figure 6.3 climbs rapidly and shows that the cache reaches full utilisation at about one hundred twenty-five concurrent requests. This saturation point arrives much sooner than the thresholds recorded for the other two models. Compared with GPT-2-XL, which saturated at about 1,000 requests, Qwen-1.5 reached its limit at roughly 130 requests, leaving a gap of 870 requests. A similarly wide difference appeared when set against DeepSeek-LLM-7B, whose memory ceiling hit around 500 requests, so Qwen-1.5 saturated nearly 370 requests earlier.

In summary, the results from this benchmark, together with the state of the art previously discussed, lead to the following findings:

- *What does higher concurrency do to the underlying infrastructure?*

  If all other factors are held fixed, raising concurrency steadily increases GPU memory pressure. With a fixed prompt and output budget, each request occupies the same slice of the KV cache, so total usage grows almost linearly with the number of active requests until the GPU memory approaches its limit. These results provide a clear picture of what is the safe operating window for each large language model, and when the memory saturation can be expected.

- *What model characteristics define the KV-cache memory ceiling?*

  The key idea is that the memory added by each token is governed by the model's attention geometry, not by the raw parameter count. Each transformer layer stores a fresh set of keys and values for the new token, so adding layers increases the per-token footprint. Within each layer, what matters is how many distinct KV sets exist and how wide they are. For example, more attention heads or a larger head dimension means more bytes per token, while choosing a lower-precision cache such as FP8 or BF16 can reduce that cost.

  Another practical constraint also shapes where the ceiling appears. When a model is deployed, its weights and runtime buffers must be loaded into GPU memory, and only the remaining headroom is available for the KV cache. A larger model therefore tends to leave less free memory even if the per-token KV size is the same, which lowers the admissible concurrency on a fixed device.

  However, it is a common misconception that the ceiling depends mainly on total parameter count. Two models with the same number of parameters can

have noticeably different concurrency limits, because components like feed-forward width, embedding size, and vocabulary size can inflate parameter count without changing the KV stored per token. What moves the ceiling is the attention-side design. For example, in multi-query attention all query heads share a single KV set per layer, and in grouped-query attention heads share KV in small groups. Both reduce the number of stored KV sets and push the ceiling to higher concurrency.

- *Why should operators care about the KV cache saturation point?*

  In production, GPUs are billed by time and not by requests. When the cache is full, the system cannot admit more concurrent work even if compute pipelines still have room. This leaves compute capacity idle while the clock keeps running. The direct result is fewer tokens delivered per GPU hour, which raises the cost per token.

  For example, if a service delivers more tokens per second in the safe region and fewer tokens per second near the memory ceiling, the dollars per million tokens increase in proportion to that drop. This means that hitting a daily token target requires more GPUs, which raises spend and lowers margin. It also means that shared fleets become less cost efficient when one workload pushes devices into the memory bound regime. This first benchmark was meant to show where this efficiency break happens for each model, which is the point where cost starts drifting upward.

AS

6.1.2 Analysis of Queue Formation Triggered by KV-Cache Ceiling

The previous benchmark established how concurrency alone drove GPU-memory pressure. However, memory percentages by themselves only indicate capacity, but they do not reveal anything about how new traffic is handled when capacity tightens. Therefore, this next section will address a simple question that matters in practice: How does approaching memory capacity change the system's ability to process new user requests?

To answer this, results from several models were examined side by side. The analysis followed four time series, namely running requests, waiting requests, cumulative completions, and KV-cache usage, to track how admission and progress changed as memory filled. The findings from each graph are discussed below.

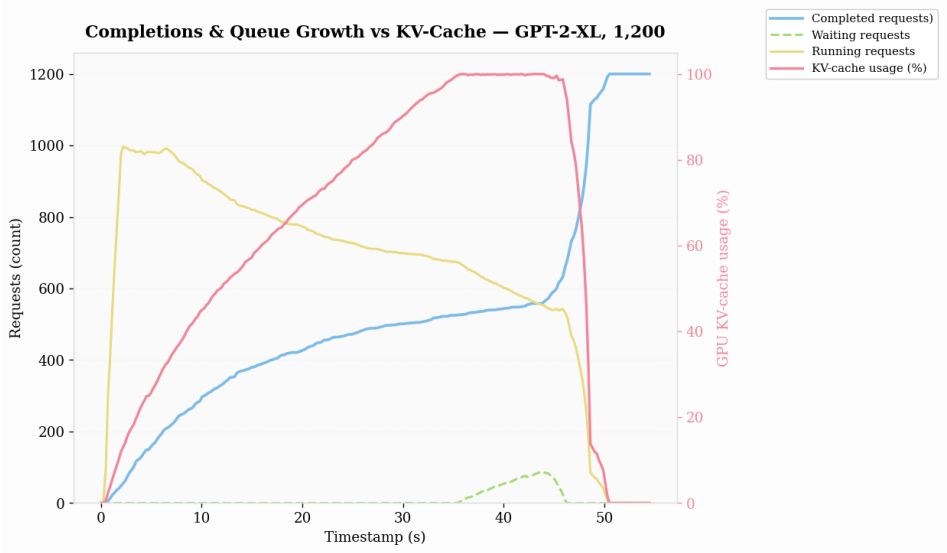


Figure 6.4: Admission, Queueing, and Memory Pressure. GPT-2-XL (1,200 concurrent user requests)

Figure 6.4 depicted the joint evolution of four time series under a burst of 1,200 requests, that the model GPT2-XL model had to process. The level of 1,200 concurrent user requests was selected for this figure because the previous benchmark (Figure 6.1) showed that it drove the KV cache to full utilisation. This made it possible to observe how the system handled both newly arriving and already running requests as capacity tightened.

As it can be observed in this first figure, most of the 1,200 requests arrived almost at once. The reason why the server admitted a very large fraction immediately was because, during prefill, each request only carried its prompt tokens and the per-request KV slice was small. This explained why the *running* curve rose to nearly one thousand within the first seconds while the *waiting* curve remained at zero. In parallel, the KV-cache percentage climbed smoothly because every active request was adding keys and values as tokens were embedded.

As decoding proceeded, each active request kept accumulating tokens, which increased the KV footprint per request. With a fixed memory pool, the system could not keep as many sequences active as during prefill. Therefore, the scheduler reduced the effective active set over time. On the plot this appeared as a gradual decline in the *running* curve from roughly one thousand toward the mid-hundreds. Additionally, cumulative completions grew steadily, which matched the expectation for fixed or capped outputs that finished at slightly different times.

A clear change in regime appeared when KV-cache usage first reached saturation,

since from that point the queue became visible. The *waiting* curve rose because additional requests could not be admitted without exceeding the KV pool. The *running* curve settled into a plateau that reflected how many sequences could fit in memory at that moment given their current token counts, called the capacity knee, in which more offered concurrency no longer translated into more active work.

Overall, this behavior was consistent with a KV cache-bound system under fixed token budgets. Early in the run, small per-request footprints allowed many sequences to run in parallel. As output tokens accumulated, per-request memory grew and the active set had to shrink. Queueing appeared only when the cache was effectively full, which indicated that memory, rather than compute, was the practical limiter of how much concurrent work this replica could sustain.

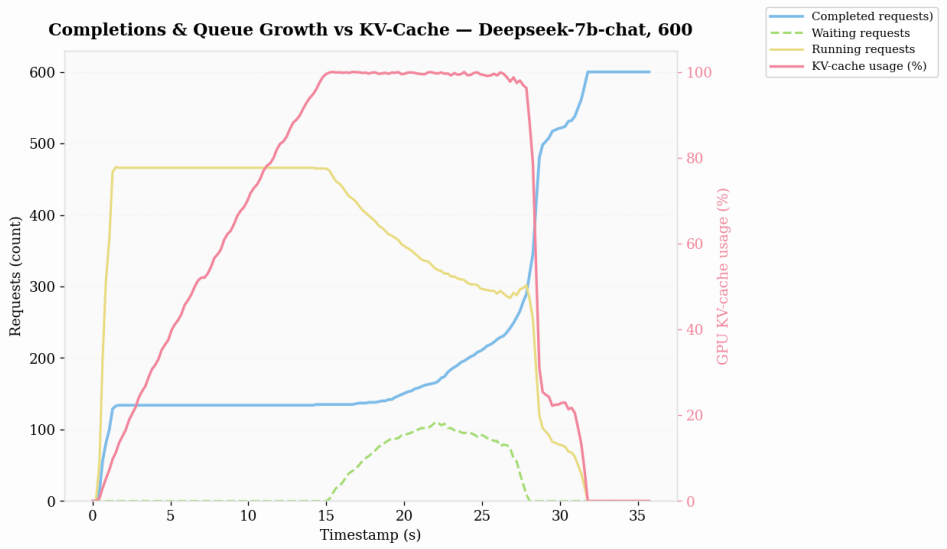


Figure 6.5: Admission, Queueing, and Memory Pressure. Deepseek-7B (600 concurrent user requests)

The same experiment was repeated for the Deepseek model in order to see how the dynamics changed across models. Figure 6.5 showed the four traces again: running requests, waiting requests, cumulative completions, and KV-cache usage.

In this model, saturation arrived much earlier and it lasted longer. Deepseek reached the KV ceiling in the low–teens seconds and then stayed pinned near 100% for many seconds. By contrast, GPT2-XL in the earlier plot approached the ceiling later and only briefly. The most direct explanation was the per-token KV size. Deepseek-7B's attention geometry, with more depth and wider heads

than GPT-2-XL, implied a larger KV footprint per token, which meant that fewer simultaneous sequences could fit before memory ran out.

A visible queue also formed mid-run. For GPT-2-XL, the queue appeared only late, while for Deepseek it rose sooner and persisted while the KV cache was pinned. However, the reason behind the formation of the queue remained the same for both models: Once the cache was full, either new arrivals or preempted sequences had to wait until enough blocks were freed.

Another main difference between the GPT2-XL model and the Deepseek model is that the running series plateaued lower. Deepseek stabilized around roughly 460 running sequences, whereas GPT-2-XL sustained close to one thousand early in the run. This was again consistent with a larger KV footprint per request in Deepseek. The engine capped the active set earlier in order to stay within the same H100 memory budget. Additionally, the completions curve for Deepseek climbed more slowly during the pinned period and then finished in a sharper wave as memory was released. GPT-2-XL showed a steadier accumulation. This pattern fit a scenario in which KV pressure and preemption delayed some decodes, so more of them completed together once headroom returned.

In short, the Deepseek figure exhibited the expected shape for a KV-bound run with a larger per-token memory footprint: an early admission surge during prefill, an early and sustained period at full KV usage, visible waiting while memory was tight, and a fast drain when many generations completed and freed blocks. The differences relative to GPT-2-XL can be attributed directly to the models' attention geometries and the resulting KV sizes.

The third model evaluated with this benchmark was Qwen1.5-14B-Chat, and its results are presented in Figure 6.6. In the case of this third model, the KV-cache filled much earlier and from a much smaller offered load. This can be proved by comparing the steepness of the KV-cache usage curve, which, with only 160 concurrent requests, climbed almost linearly and hit 100% around 16–17 s, then stayed pinned until 23–24 s. The reason for this steep rise in GPU memory utilization is that Qwen-14B's attention geometry allocated more bytes per token than the 1.5 B and 7 B models, so the same number of simultaneous requests consumed memory faster and left less headroom.

Even with a rapidly growing memory utilization, the system still admitted all 160 requests immediately, as the yellow *running* line jumped straight to 160 at the start and remained there for many seconds. Nonetheless, as decoding progressed, each request's KV slice grew, so the fixed memory pool could no longer support all 160 at once and the running line started to step down while KV stayed saturated.

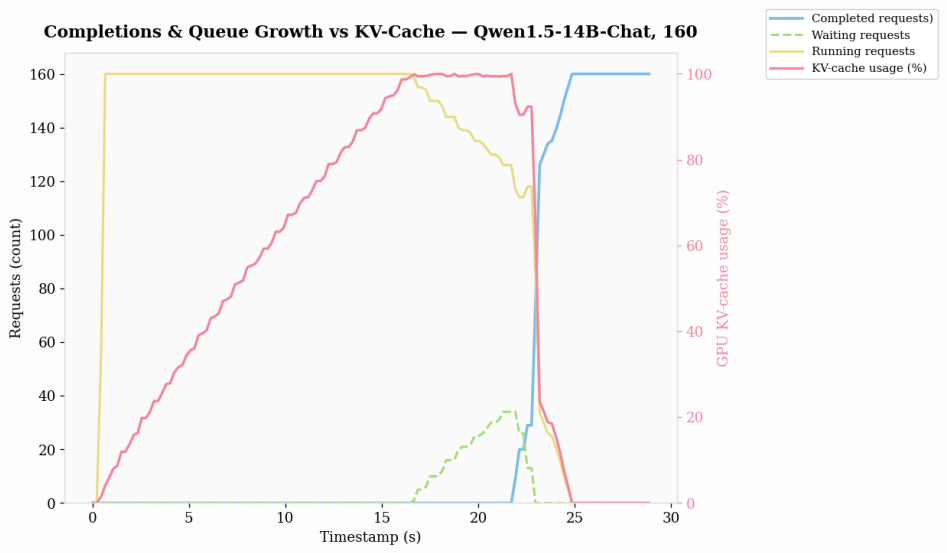


Figure 6.6: Admission, Queueing, and Memory Pressure. Qwen1.5-14B-Chat (160 concurrent user requests)

The most distinct trait between the results for Qwen1.5-14B and the previous models was how no requests were completely finished until the very end of the run, at about 22 s in. Instead, requests progressed slowly while the cache was saturated and then finished close together once the first wave completed and freed blocks. GPT-2-XL showed a steadier accumulation of completions, while Deepseek sat in between, with some mid-run completions but still a late surge. The Qwen shape is what one expects when memory is binding for most of the run and the scheduler must ration active decoding.

Taken together, the Qwen plot is the "most memory-bound" of the three. The underlying reason was the larger KV bytes per token implied by Qwen-14B's attention geometry. With fixed token budgets and identical hardware, that larger per-token footprint makes concurrency translate into memory pressure faster, which in turn changes the dynamics of running, waiting, and completion compared with GPT-2-XL and Deepseek-7B.

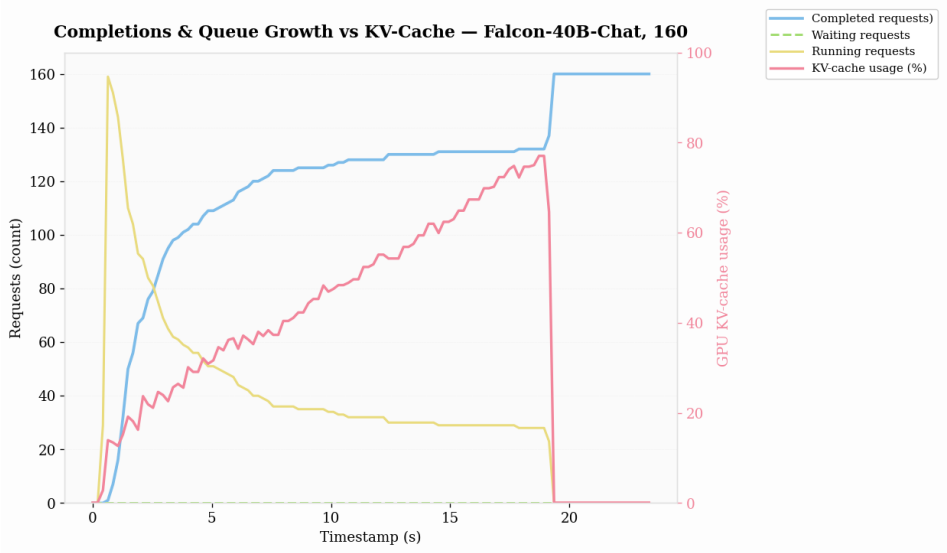


Figure 6.7: Admission, Queueing, and Memory Pressure. Falcon-40B (160 concurrent user requests)

This final experiment was also conducted with a fourth model, Falcon-40B. Because Falcon had the largest parameter count, the initial expectation was to see a stronger version of the previous runs, since loading more weights leaves less headroom for KV and a larger model might increase the memory cost per token. However, Figure 6.7 showed a different outcome and invited a closer look.

First, no visible queue appeared at any time, since the green *waiting* series stayed at zero while all 160 requests completed. This contrasted with GPT-2-XL, DeepSeek-7B, and Qwen-14B, where a queue emerged once the KV cache reached 100%. The most direct explanation was that Falcon never drove the KV cache to its ceiling under this load, so admission control did not need to hold new work back.

Two architectural facts explained why KV usage remained below saturation. Falcon-40B used multi-query attention, where all query heads in a layer shared a single KV set. In standard multi-head attention each head keeps its own KV, which makes the per-token KV bytes grow with the number of heads. With multi-query attention the model kept one KV per layer instead of one per head, which reduced the per-token KV footprint. As a result, even though the model was larger in total parameters, the part that governs KV growth per token was lighter. With fewer bytes per token, the same 160 in-flight requests consumed less GPU memory, so the cache did not fill.

The *running* series also followed a different shape. It jumped to nearly all 160 at the beginning because prefill could embed many prompts in parallel while their KV

slices were still small. It then declined to a stable plateau as decoding progressed, since each active request accumulated tokens and its KV grew. vLLM's chunked prefill and scheduler kept only as many sequences decoding as comfortably fit in memory, which maintained a steady band of *running* requests. Because KV usage stayed below the ceiling, admission did not block, and the *waiting* series remained at zero while the active set was managed internally by the scheduler.

Another interesting insight is how completions accumulated smoothly from start to finish. The blue curve rose almost monotonically to 160, without the mid-run stall observed when other models sat at 100% KV usage. This behavior fit the picture above, since there was no admission backlog and the decode set stayed stable.

To summarize, Falcon-40B behaved differently because its per-token KV cost was lower and the scheduler kept decode concurrency within that lighter budget. The run was not KV-cache-bound at 160 offered requests, which produced no queue, sub-saturation KV usage, a shrinking *running* set after prefill, and a steady march of completions. This reinforced that KV geometry, rather than parameter count alone, determined whether a given workload would reach the memory ceiling.

With the results in view, the analysis can be distilled into a small set of insights that matter in practice, outlined below:

- *How does approaching memory capacity change the system's ability to process new user requests?*

  The figures presented in this subsection showed a simple pattern that matters in practice. While KV usage was low, new traffic was admitted immediately, running stayed high, and waiting stayed at zero. As decoding progressed, each request grew its KV slice, which meant the active set had to shrink even if more users arrived. Near the top of the KV scale the system stopped turning offered load into active work, running flattened, waiting appeared, and completions started to bunch when memory was freed. In other words, once memory was tight, adding more concurrency did not produce more useful work. This naturally leads to the next question, which is how to choose a model that avoids hitting this knee at the system's target SLOs.

- *How should a model be selected for a given SLO when two options have similar parameter counts but different per-token KV footprints?*

  The comparison across models made the choice clear. What mattered for admission at target concurrency was the per-token KV footprint, not the raw parameter count. For example, Qwen-14B saturated early because each token occupied more KV bytes, while Falcon-40B stayed below the ceiling at

the same offered load because its attention design shared KV and reduced bytes per token.

As a practical recommendation, to select a model that stays admission-friendly at peak concurrency, preference should be given to architectures with smaller KV per token or to using lower-precision KV, provided quality targets are still met.

- *What level of headroom keeps the service inside a no-queue, predictable regime?*

  The figures indicated that visible waiting began only when KV lived in the high nineties, and that stability returned as soon as a small amount of memory was freed.

  As a practical operating rule, to keep the service in a safe regime, KV usage should be tracked continuously because it is the first signal to move when memory becomes the limiter. The system should be held inside a target band below the knee by capping tokens in flight, since this directly controls how much KV is allocated at once. In these runs, leaving about ten to fifteen percent headroom kept admission smooth. The exact margin should be revalidated whenever model geometry, precision, or workload mix changes, because each of these shifts the bytes per token or the baseline weight footprint.

With admission dynamics now quantified, the next step is to ask what this means for users and SLOs. The following subsections therefore examine how approaching the memory ceiling changes aggregate tokens per second that the system can provide, as well as the latency that users experience as a result of it.

6.1.3 Analysis of KV-Cache Saturation Effects on Aggregate Throughput

After examining the earlier figures, the next question that naturally arises is how memory saturation and queuing shape the performance of an inference system, and in particular, what this means for both inference service providers and end users. While the previous results highlighted how resources are consumed under different concurrency levels, it remains to be understood how these pressures ultimately translate into the metrics that companies and end users directly perceive.

To address this, an analysis was carried out, whose objective was to capture how throughput evolved as the KV-cache approached its limits and began to saturate. This analysis was repeated across different models in order to assess whether the choice of model influences throughput behavior or alters the severity of degradation once saturation occurs.

The results of these experiments are summarized in the figures that follow, which trace throughput as a function of elapsed time under varying concurrency levels. These will be discussed in detail below.

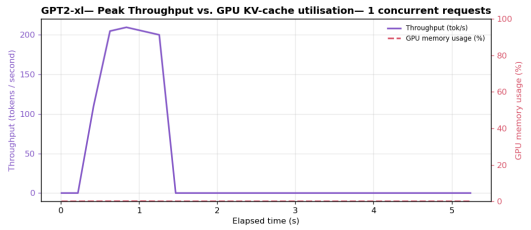


Figure 6.8: Throughput over time for GPT-2-XL-1.5B with 1 concurrent request.

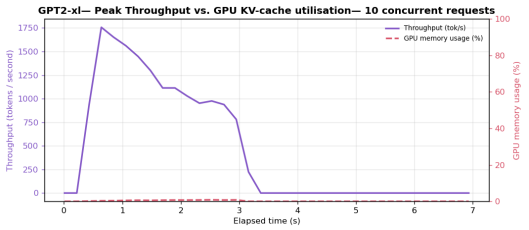


Figure 6.9: Throughput over time for GPT-2-XL-1.5B with 10 concurrent requests.

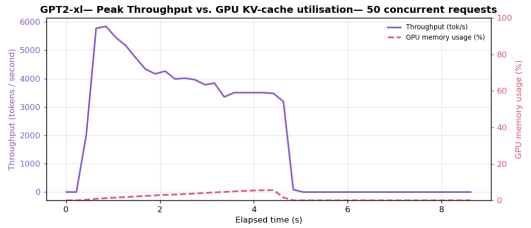


Figure 6.10: Throughput over time for GPT-2-XL-1.5B with 50 concurrent requests.

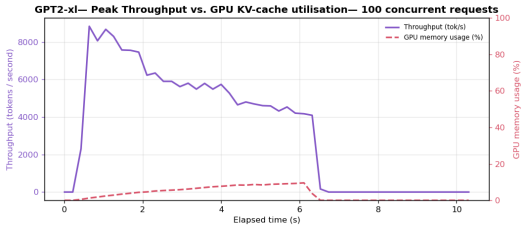


Figure 6.11: Throughput over time for GPT-2-XL-1.5B with 100 concurrent requests.

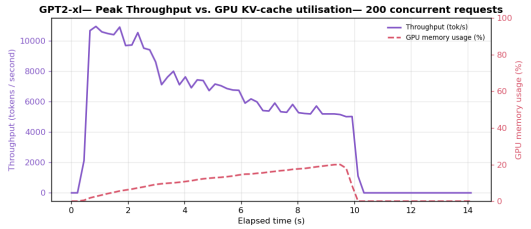


Figure 6.12: Throughput over time for GPT-2-XL-1.5B with 200 concurrent requests.

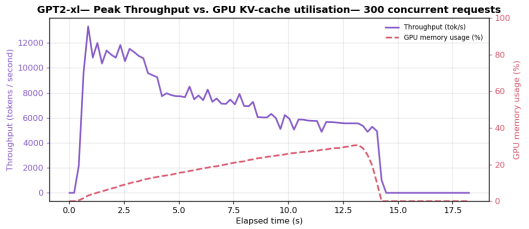


Figure 6.13: Throughput over time for GPT-2-XL-1.5B with 300 concurrent requests.

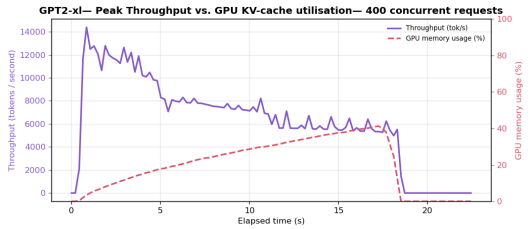


Figure 6.14: Throughput over time for GPT-2-XL-1.5B with 400 concurrent requests.

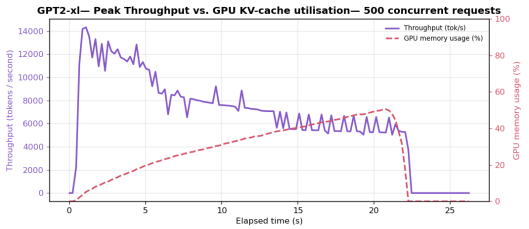


Figure 6.15: Throughput over time for GPT-2-XL-1.5B with 500 concurrent requests.

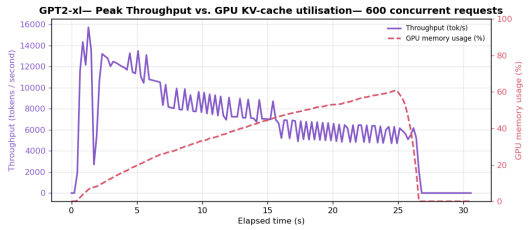


Figure 6.16: Throughput over time for GPT-2-XL-1.5B with 600 concurrent requests.

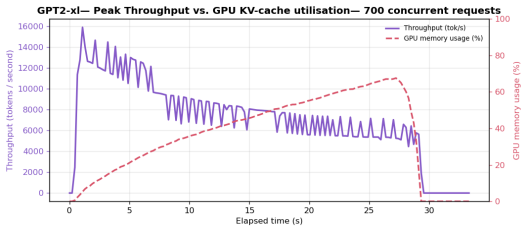


Figure 6.17: Throughput over time for GPT-2-XL-1.5B with 700 concurrent requests.

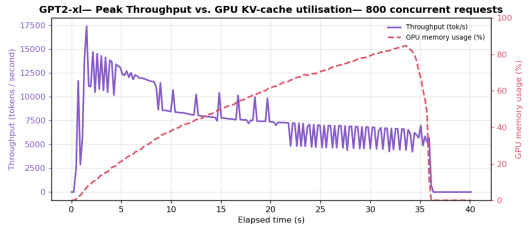


Figure 6.18: Throughput over time for GPT-2-XL-1.5B with 800 concurrent requests.

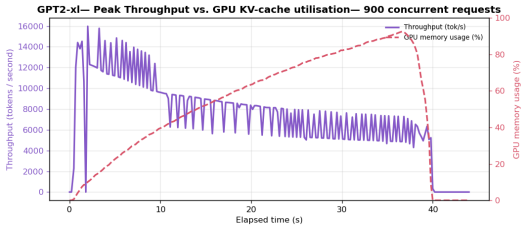


Figure 6.19: Throughput over time for GPT-2-XL-1.5B with 900 concurrent requests.

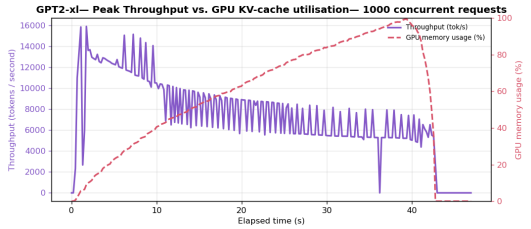


Figure 6.20: Throughput over time for GPT-2-XL-1.5B with 1000 concurrent requests.

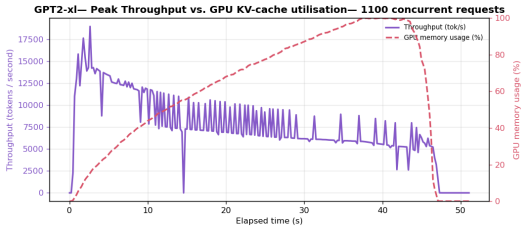


Figure 6.21: Throughput over time for GPT-2-XL-1.5B with 1100 concurrent requests.

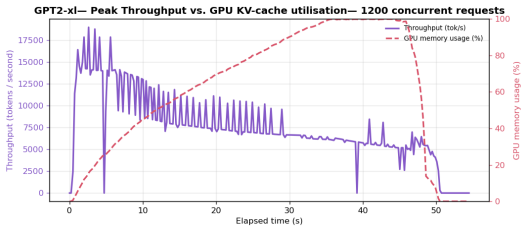


Figure 6.22: Throughput over time for GPT-2-XL-1.5B with 1200 concurrent requests.

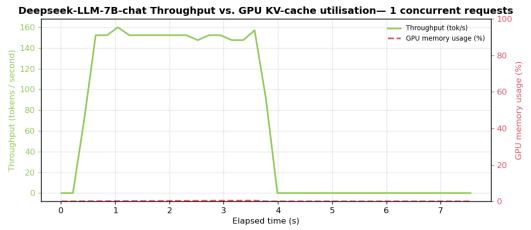


Figure 6.23: Throughput over time for DeepSeek with 1 concurrent request.

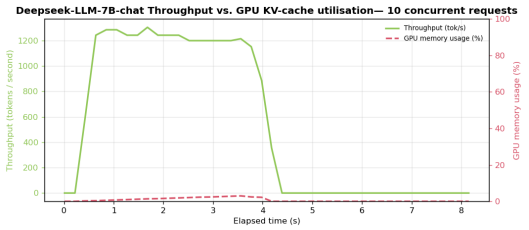


Figure 6.24: Throughput over time for DeepSeek with 10 concurrent requests.

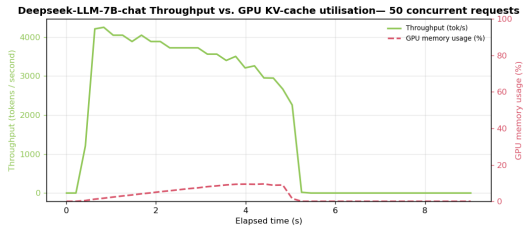


Figure 6.25: Throughput over time for DeepSeek with 50 concurrent requests.

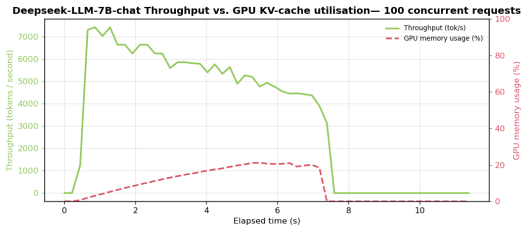


Figure 6.26: Throughput over time for DeepSeek with 100 concurrent requests.

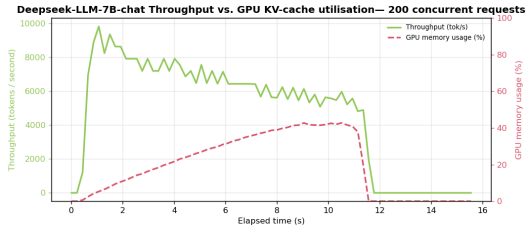


Figure 6.27: Throughput over time for DeepSeek with 200 concurrent requests.

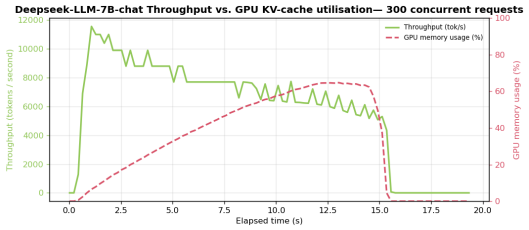


Figure 6.28: Throughput over time for DeepSeek with 300 concurrent requests.

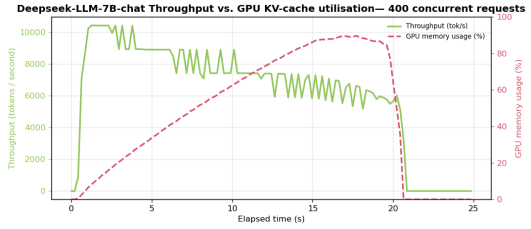


Figure 6.29: Throughput over time for DeepSeek with 400 concurrent requests.

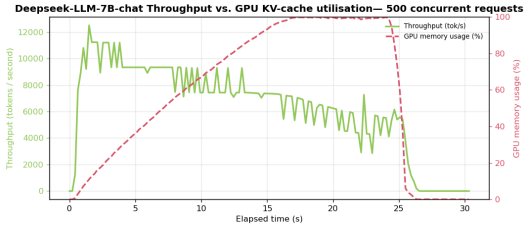


Figure 6.30: Throughput over time for DeepSeek with 500 concurrent requests.

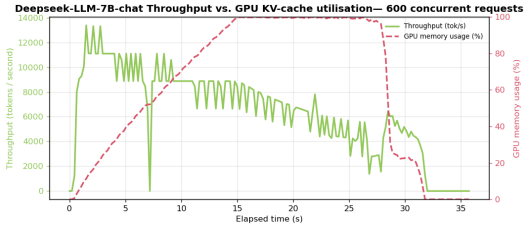


Figure 6.31: Throughput over time for DeepSeek with 600 concurrent requests.

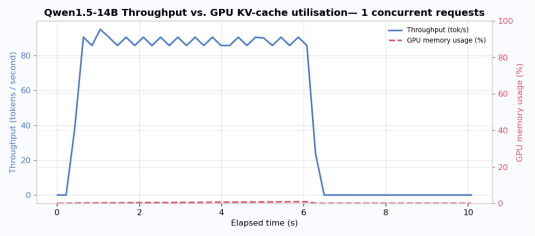


Figure 6.32: Throughput over time for Qwen-14B with 1 concurrent request.

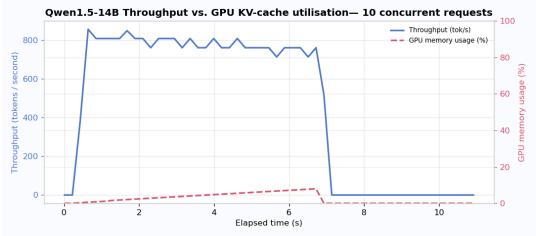


Figure 6.33: Throughput over time for Qwen-14B 10 concurrent requests.

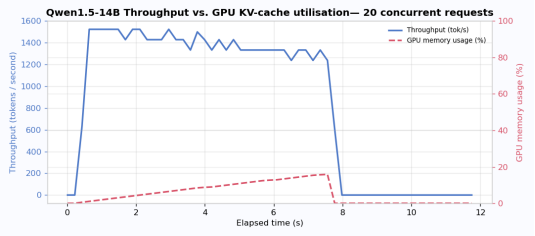


Figure 6.34: Throughput over time for Qwen-14B 20 concurrent requests.

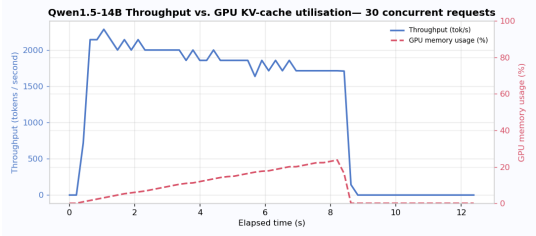


Figure 6.35: Throughput over time for Qwen-14B 30 concurrent requests.

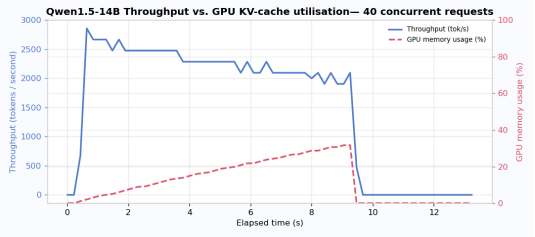


Figure 6.36: Throughput over time for Qwen-14B 40 concurrent requests.

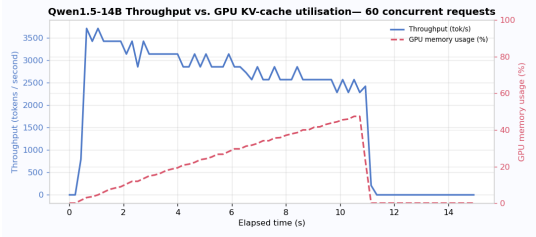


Figure 6.37: Throughput over time for Qwen-14B 60 concurrent requests.

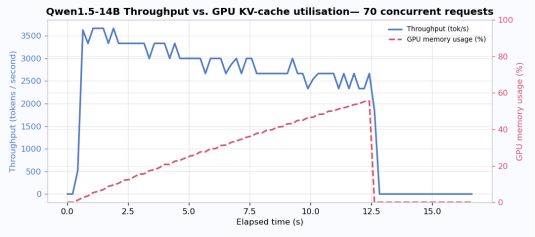


Figure 6.38: Throughput over time for Qwen-14B 70 concurrent requests.

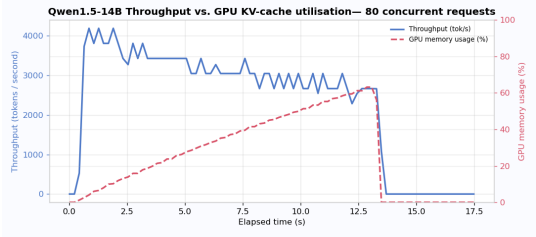


Figure 6.39: Throughput over time for Qwen-14B 80 concurrent requests.

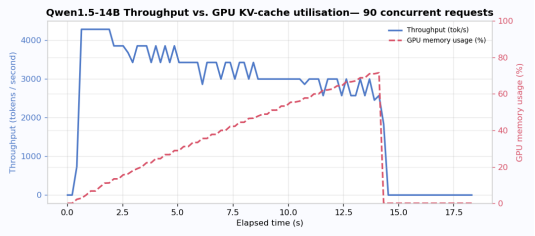


Figure 6.40: Throughput over time for Qwen-14B 90 concurrent requests.

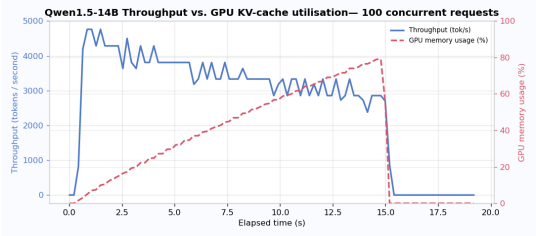


Figure 6.41: Throughput over time for Qwen-14B 100 concurrent requests.

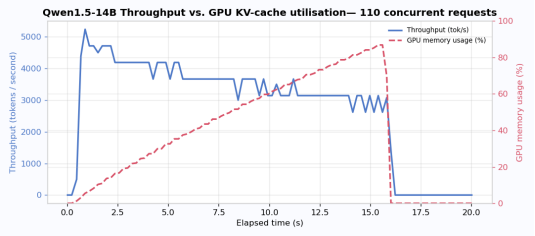


Figure 6.42: Throughput over time for Qwen-14B 110 concurrent requests.

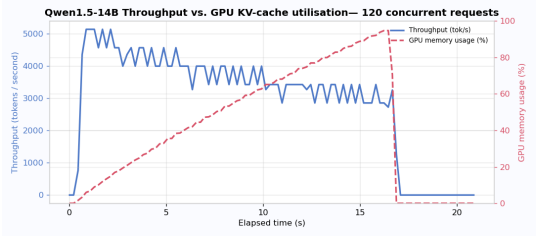


Figure 6.43: Throughput over time for Qwen-14B 120 concurrent requests.

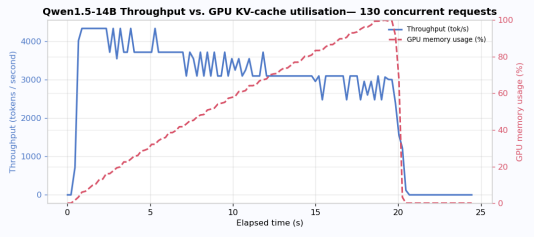


Figure 6.44: Throughput over time for Qwen-14B 130 concurrent requests.

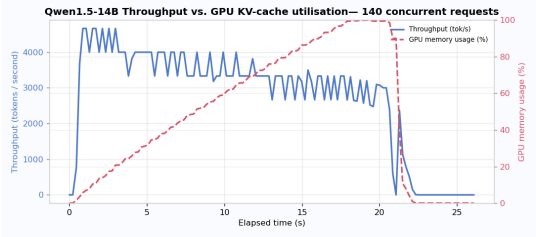


Figure 6.45: Throughput over time for Qwen-14B 140 concurrent requests.

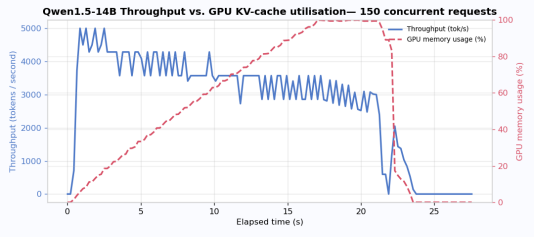


Figure 6.46: Throughput over time for Qwen-14B 150 concurrent requests.

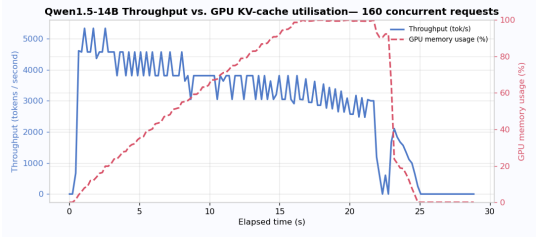


Figure 6.47: Throughput over time for Qwen-14B 160 concurrent requests.

These figures reveiled a repetitive pattern, since although the exact throughput val-

ues varied across models, the overall shape of the curves remained consistent.

One of the most important observations gathered from the figures was that throughput tended to decline over time, even before the KV-cache was fully saturated. At first glance, one might expect throughput to remain stable until memory exhaustion occurs, but this was not the case. Instead, a gradual erosion was seen across all runs. This decline is linked to the fact that as more sequences accumulated in memory, scheduling and memory-access patterns became increasingly complex. For example, each additional sequence required bookkeeping and introduced contention in memory access, so the system spent slightly more time managing state and slightly less time generating new tokens.

The implication of this pattern for inference providers is that performance degradation begins well before the "cliff" of memory exhaustion, meaning that running a system near its limit is risky even if it has not yet hit the ceiling.

The second major observation came from runs where the KV-cache eventually saturated. In these cases, throughput did not simply plateau at a fixed maximum value. Instead, it decreased steadily over time, eroding in a roughly linear fashion once the cache was full. This collapse was driven by several compounding factors. Once the KV cache was full, overhead rose because the system had to evict and remap memory pages in order to admit new tokens. This process slowed generation since some sequences were delayed or cut short, and GPU kernels faced greater contention while waiting for memory operations to complete. As a result, throughput declined rather than stabilizing, showing that saturation did not just cap performance but actively degraded it.

Together, these results show that saturation destabilizes the inference system. This is due to the fact that, once the cache is full, every new request makes the system less efficient. For providers, this highlights the importance of carefully managing load to avoid not just the memory ceiling itself but also the performance cliff that follows. For end users, it means that latency and quality of service can deteriorate more sharply than expected once the system is pushed beyond safe utilization levels.

To conclude, the analysis of the presented figures revealed the following insights, summarized as the following points:

- *How does throughput evolve as concurrency increases, and what happens once memory saturation is reached?*

  Throughput declined over time even before memory saturation, reflecting the growing KV footprint of active sequences during decoding. Once the KV-cache reached its limit, the decline became sharper, showing that saturation

not only prevented further scaling but actively eroded system efficiency.

- *Why does flattening throughput at the memory ceiling matter for inference providers in terms of efficiency and cost per token?*

  When many user requests are processed at the same time, the system initially produces tokens at a higher rate, which is reflected as higher throughput. However, this growth does not continue indefinitely. Once the memory reserved for storing intermediate results is fully used, the system cannot admit more active work, so throughput stops increasing and instead flattens at a ceiling. From that point onward, no matter how much additional demand is offered, the number of tokens completed per second remains roughly constant, or even diminishes as observed in the previous figures.

  This plateau is important because it means that the hardware is not producing more useful work even though it is being asked to handle more requests. In practice, GPUs continue to spend time maintaining the waiting queue, holding partial states, and scheduling work that cannot immediately run. This overhead consumes the same amount of costly GPU hours without delivering extra completions. For an inference provider, this translates directly into efficiency loss, because the cost per generated token rises once the system is past the ceiling.

  Another way to see this is by looking at the business side. Providers often operate under fixed budgets for compute, so every second of GPU time that does not translate into more output increases the average cost of serving each request. When throughput flattens, the system is no longer scaling with demand, and the service ends up charging the same resources for less incremental benefit. This is why the ceiling is not only a technical detail but also a financial one: it marks the boundary where extra load turns into waste rather than more revenue or more satisfied users.

  In short, flattening throughput signals that the system has reached the point where memory constraints, rather than raw compute power, are in control. For operators, this means that keeping the service inside a safe headroom below the ceiling is not only a way to ensure smoother performance but also a way to make sure that every unit of GPU time is used effectively and that cost per token remains predictable.

- *How does sustained throughput relate to predictable service-level objectives, and what risks arise when queues form?*

  The key ideas is that while throughput on its own is a useful number, its real importance comes from how stable it remains over time. A provider that

promises to deliver results within a certain time window, such as under a service-level objective (SLO), depends on throughput staying consistent. If the number of tokens produced per second holds somewhat steady, then the system can reliably predict how long it will take to serve a given volume of requests. This predictability is what makes SLOs enforceable, since guarantees about response times rest on the assumption that work flows through the system at a steady pace.

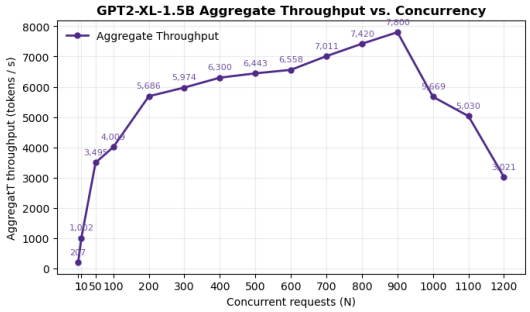
The risk appears when memory becomes saturated and a queue begins to form. At that moment, throughput may go flat and even decrease at the ceiling, but requests in the queue no longer progress immediately. Therefore, each new user entering the system has to wait until enough memory is freed by completed requests, which results in a rise in latency experienced by individual users. From a business perspective, this is dangerous, because the service is still incurring the same costs to run the hardware, yet users perceive a slowdown and may see SLOs being violated.

Therefore, sustained throughput without queues is a sign of a healthy operating regime where efficiency and predictability align. Once queues appear, the same hardware is no longer translating directly into satisfied users, since part of its time is spent holding work that cannot advance. This highlights why inference providers must not only measure throughput but also monitor admission and queueing closely. It is the combination of both that determines whether SLOs are being met in practice.

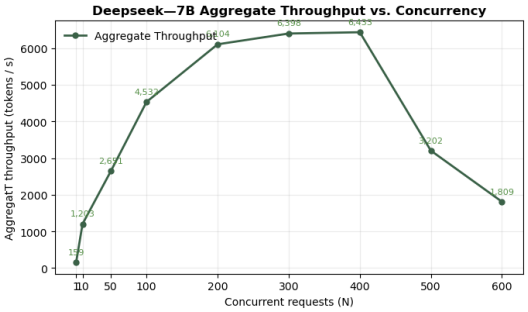
6.1.4 Analysis of Peak Throughput under Varying Concurrency

After examining how throughput evolved over time at fixed levels of concurrency, the next step was to analyze the problem from a different perspective. Instead of observing the full trajectory of each run, this section focuses on the maximum throughput achieved at each concurrency level. In other words, each point in the plots below represents the highest throughput observed during the run for a given number of concurrent requests, regardless of when it occurred.

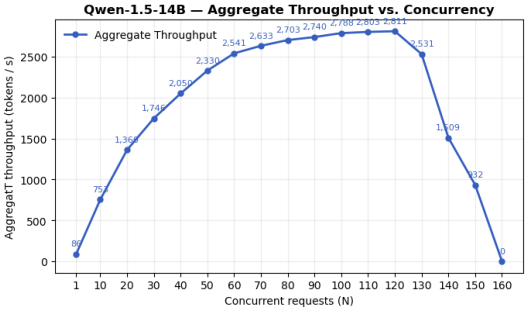
The following figures present the results for the different models evaluated. Despite differences in size and configuration, all of them exhibited a consistent overall shape: throughput first grew almost linearly with concurrency, then entered a regime of slower improvement, and finally declined once the system was overloaded.



(a) Concurrency threshold for KV-cache saturation (GPT2-xl-1.5B)



(b) Concurrency threshold for KV-cache saturation (Deepseek-7B)



(c) Concurrency threshold for KV-cache saturation (Qwen1.5-14B)

Figure 6.48: Concurrency threshold for KV-cache saturation as a function of model size across different models.

Several key insights emerged from this analysis:

- *How does peak throughput evolve with increasing concurrency?*

  At first, throughput rose nearly in proportion to the number of concurrent requests, since additional demand could be absorbed without contention. Once the hardware was fully engaged, however, each new request still contributed some additional throughput, but at a slower rate, as resources had to be shared. Finally, when concurrency became excessive, throughput began to decline because the overhead of coordination and contention outweighed the useful work contributed by new requests.

- *What does this reveal about inference system behavior?*

  The results show that systems naturally progress through three performance phases as concurrency grows: an initial period of efficient scaling, a zone of

diminishing returns once resources are saturated, and an eventual decline under overload. This pattern highlights the trade-off between maximizing utilization and maintaining stability, as the most efficient operating point lies between underuse and overload.

- *Are these patterns model-dependent?*

  While the absolute concurrency thresholds varied across models, the qualitative shape of the curve remained consistent. This indicates that the observed progression is not an artifact of a particular model, but instead a general property of LLM inference workloads.

Having seen how throughput declined as memory pressure grew, the next step is to ask how this behavior translates into what users actually notice. The following subsection therefore explores this question by examining how the collapse of throughput at saturation reshapes the latency that users experience, and how that can interfere with established SLOs.

6.1.5 Analysis of Latency Degradation Under Increasing Concurrency

This subsection examines how latency evolved as the number of concurrent requests increased. The goal was to identify the extent to which scaling concurrency impacts responsiveness, particularly in terms of time-to-first-token (TTFT). The following figures present the results for the four tested models: GPT2, Deepseek, Qwen, and Falcon.

In theory, what should be observed is a "flat–then–knee–then–steep" curve in TTFT as concurrency grows. At low concurrency levels, latency remains stable, followed by a sudden inflection point once the system approaches saturation, and finally a sharp rise in delay as resources become overloaded.

However, not all models under study exhibited this expected behavior. In several cases, the results showed a near-linear increase in TTFT as concurrency rose. This can be attributed to the workload generator design. Because the driver submitted all N requests simultaneously in burst mode, TTFT measurement effectively incorporated the waiting time in the submission queue. As a result, each additional request waited behind the others, producing a linear growth with slope approximately equal to the prefill time per request. In this scenario, the concurrency experiment reflects serialized prefill processing rather than steady-state queuing effects, explaining the deviation from the theoretical knee-shaped curve.

The main conclusions from this experiment can be summarized as follows:

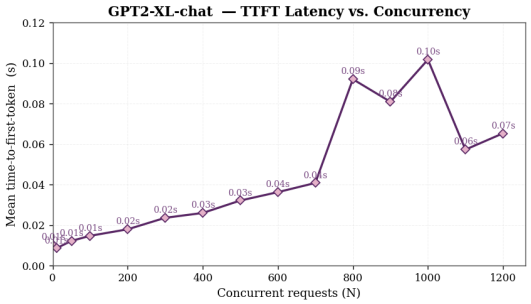


Figure 6.49: TTFT vs. concurrency for GPT2-XL-chat.

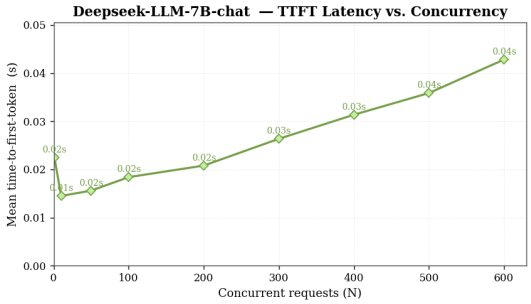


Figure 6.50: TTFT vs. concurrency for Deepseek-LLM-7B-chat.

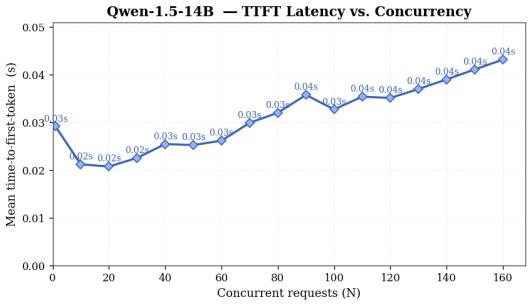


Figure 6.51: TTFT vs. concurrency for Qwen-1.5-14B.

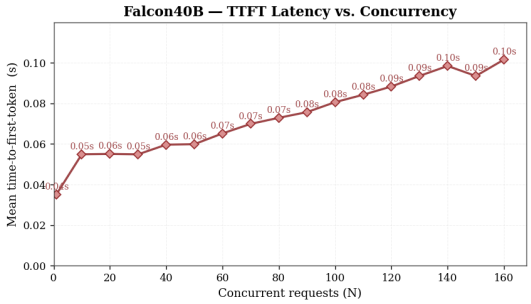


Figure 6.52: TTFT vs. concurrency for Falcon40B.

- *How does TTFT evolve under increasing concurrency?* For some models, TTFT increased in a nearly linear fashion with concurrency, rather than remaining flat at first and then rising sharply. This indicates that the workload generator's burst submission amplified queueing effects at every concurrency level.
- *What does this reveal about system behavior?* The discrepancy between the expected knee-shaped curve and the observed linear trend highlights the sensitivity of latency measurements to workload design. In burst scenarios, TTFT reflects serialized prefill delays, while in sustained arrival scenarios, batching and scheduling effects would dominate. This underlines the importance of workload configuration when interpreting latency results.

6.1.6 Analysis of Model Size on Concurrency Limits

Building on the observations of Subsection 6.1.1, where GPU memory utilisation was tracked across rising concurrency levels, the next benchmark investigated how

the size of the language model itself constrained concurrency. By keeping the hardware configuration constant and varying only the parameter count and architecture, the test aimed to identify the point at which different models first exhausted the KV cache and could no longer admit additional user requests.

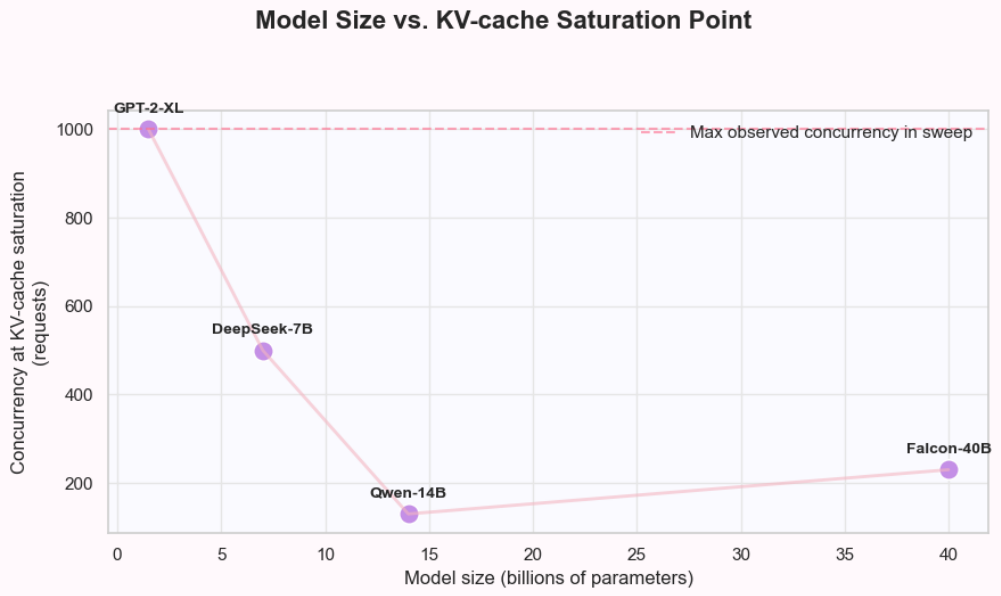


Figure 6.53: Concurrency threshold for KV-cache saturation as a function of model size

Figure 6.53 plots each model's parameter count on the horizontal axis against the concurrency level at which its KV cache became full. GPT-2-XL, with 1.5 billion parameters, did not saturate until the benchmark reached roughly 1 000 simultaneous requests. DeepSeek-LLM-7B, at 7 billion parameters, saturated earlier, near 500 requests. Qwen-1.5-14B, with 14 billion parameters, reached its ceiling at about 130 requests. Finally, Falcon-40B, despite having nearly three times more parameters than Qwen, saturated at around 200 concurrent requests.

The first three points traced a steep downward slope, suggesting that larger models impose heavier memory demands per request and therefore reduce the number of concurrent users a single GPU can support. However, the Falcon-40B result highlighted that model size alone does not fully determine concurrency limits. Falcon employs multi-query attention (MQA), which shares KV-cache entries across heads and thus moderates memory growth compared to standard multi-head attention. Even so, its concurrency ceiling remained lower than Qwen-14B, showing that architecture-level choices shape how efficiently memory is used and may counterbalance or exacerbate the raw effect of parameter count.

The figure therefore illustrates that while memory rather than compute becomes the dominant bottleneck as models scale, the exact concurrency limit depends not only on parameter size but also on design decisions such as the attention mechanism. Therefore, capacity planning needs to consider both model size and architecture, since features like MQA can change how many users a model can support, even if the parameter count stays the same.

6.2 Analysis of Prompt and Generation Sequence Lengths on Inference Latency

6.2.1 Prefill Only

The objective of this benchmark was to evaluate how increasing the prompt length provided by a user impacts overall inference latency. To obtain a sufficient number of data points capturing scaling behavior, the experiment was conducted with the Mistral-7B model, which supports a context length of up to 128k tokens. This extended context window enabled long runs and allowed the effect of prompt length on system performance to be quantified with precision.

Selected results from this benchmark are presented in the following figures:

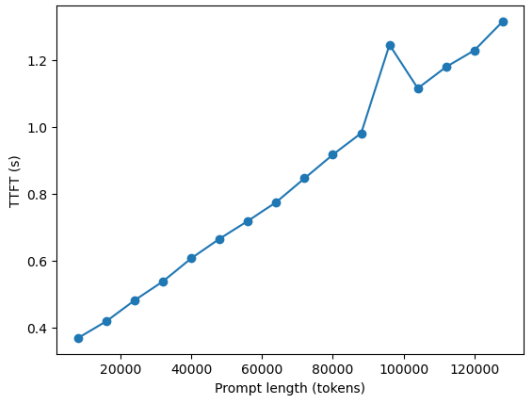


Figure 6.54: TTFT vs. prompt length for Mistral-7B-128k.

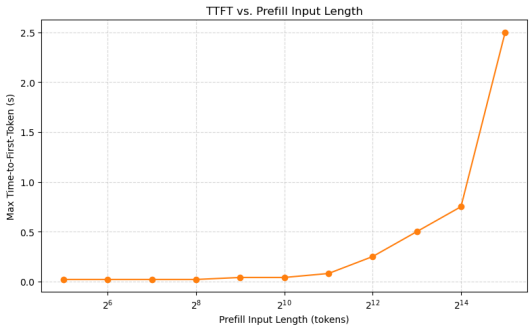


Figure 6.55: Max TTFT vs. prefill input length (Qwen, prefill-only benchmark).

Based on the analysis of these figures, the main conclusions are as follows:

- *How does TTFT scale with prompt length?*

  TTFT increased approximately linearly as prompt length grew. This behavior reflects the fact that each additional token requires a full forward pass

through all transformer layers, making the compute cost per token effectively constant. The slope of this linear growth depends on model size, since larger models perform more FLOPs per token and therefore exhibit steeper increases.

- *Why is TTFT nonzero even for very short prompts?*

  Even at minimal input lengths, TTFT did not fall to zero. This is explained by fixed runtime overheads such as framework initialization, batching windows, scheduling, and kernel setup. As a result, the plotted curves showed a positive intercept at zero tokens.

6.3 Analysis of Intra-Node Scaling Effects on Inference Performance

A natural extension of the previous experiments was to ask how inference performance changes when a model is distributed across multiple GPUs within the same node. The motivation for this family of benchmarks lies in evaluating whether tensor parallelism can effectively raise throughput ceilings, delay memory saturation, or otherwise alter the efficiency profile of the system. The central question is whether splitting the model across two devices improves performance in practice, or if the added coordination overhead offsets the potential gains.

To explore this, multiple tests were conducted in which a fixed number of concurrent user requests were issued to the inference system under two deployment configurations: first, with the model loaded entirely on a single GPU (`TP=1`); and second, with the model split evenly across two GPUs (`TP=2`). For this study, the chosen workload was `LLaMA-70B-FP8`, a model that occupied roughly half of the available memory when placed on a single GPU. This ensured that the model could both fit comfortably on one device and also be meaningfully distributed across two, without either configuration being artificially constrained.

The results obtained from these experiments are presented in the following figures:

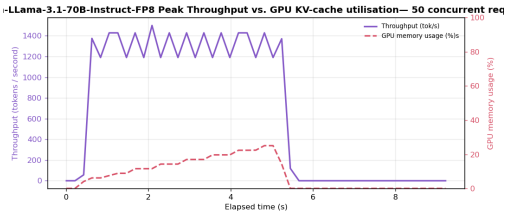


Figure 6.56: Throughput vs. KV-cache utilisation (Tensor Parallel = 1, Llama-3.1-70B-FP8, 50req)

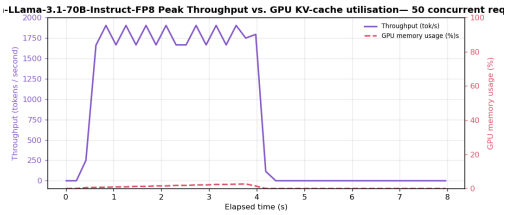


Figure 6.57: Throughput vs. KV-cache utilisation (Tensor Parallel = 2, Llama-3.1-70B-FP8, 50req)

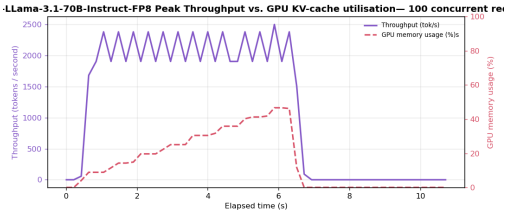


Figure 6.58: mThroughput vs. KV-cache utilisation (Tensor Parallel = 1, Llama-3.1-70B-FP8, 100req)

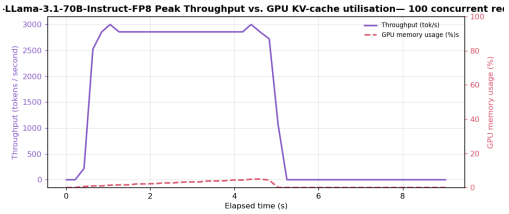


Figure 6.59: Throughput vs. KV-cache utilisation (Tensor Parallel = 2, Llama-3.1-70B-FP8, 100req)

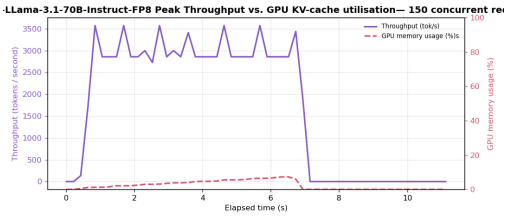


Figure 6.60: Throughput vs. KV-cache utilisation (Tensor Parallel = 1, Llama-3.1-70B-FP8, 150req)

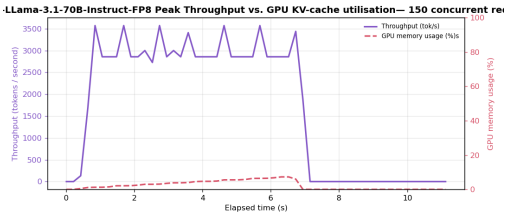


Figure 6.61: Throughput vs. KV-cache utilisation (Tensor Parallel = 2, Llama-3.1-70B-FP8, 150req)

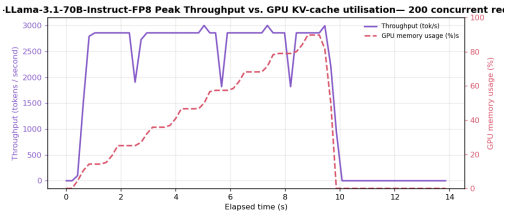


Figure 6.62: Throughput vs. KV-cache utilisation (Tensor Parallel = 1, Llama-3.1-70B-FP8, 200req)

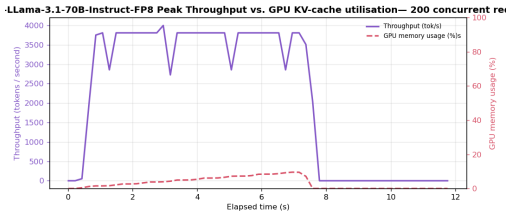


Figure 6.63: Throughput vs. KV-cache utilisation (Tensor Parallel = 2, Llama-3.1-70B-FP8, 200req)

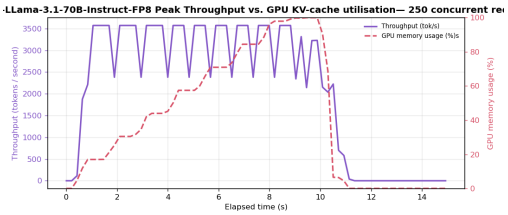


Figure 6.64: Throughput vs. KV-cache utilisation (Tensor Parallel = 1, Llama-3.1-70B-FP8, 250req)

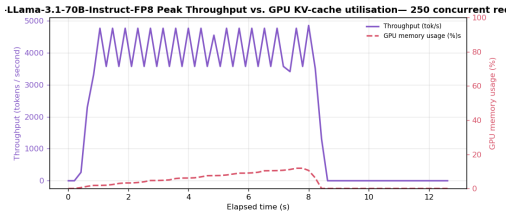


Figure 6.65: Throughput vs. KV-cache utilisation (Tensor Parallel = 2, Llama-3.1-70B-FP8, 250req)

The following results can be summarized into these insights:

- *How does increasing tensor parallelism from 1 to 2 change the throughput curve as concurrency scales?*

  Across all tested levels of concurrency, throughput increased when moving from `TP=1` to `TP=2`. The magnitude of this improvement, however, depended strongly on the system load. At low concurrency, the difference between the two configurations remained minor, since a single GPU could already process requests efficiently. As concurrency grew, the single-GPU case struggled to sustain performance, while the two-GPU configuration benefited from the additional compute and memory resources, resulting in more pronounced gains.

- *Does* `TP=2` *delay the point of KV-cache saturation compared to* `TP=1`*, and by how much?*

  The introduction of a second GPU shifted the saturation point to higher concurrency levels. With tensor parallelism, the effective KV-cache capacity doubled, reducing memory pressure and allowing the system to accommodate more requests before reaching saturation.

- *At what concurrency levels do the benefits of tensor parallelism become most visible?*

  The benefits of `TP=2` became most apparent at high concurrency, where the single-GPU configuration exhibited clear throughput degradation. In these scenarios, distributing the model across two devices prevented bottlenecks and delivered higher sustained throughput.

- *Are throughput fluctuations (peaks and valleys) more pronounced under* `TP=2`*, and what does this suggest about coordination overhead?*

  Throughput fluctuations were more pronounced in the `TP=1` runs, reflecting the strain placed on a single device as it approached its limits. Under `TP=2`, the additional resources provided a more stable execution environment, indicating that coordination overhead did not outweigh the advantages of parallelization.

6.4 Applied Recommendations Informed by Empirical Findings

Following the successful implementation of the automated benchmarking pipeline and the multivariable analysis of inference performance conducted in the previous sections, this final part presents a set of concrete recommendations derived from the key empirical insights gathered throughout the study. These recommendations are structured around four core roles involved in the design and operation of LLM inference infrastructure: developers, system architects, infrastructure providers, and hardware manufacturers.

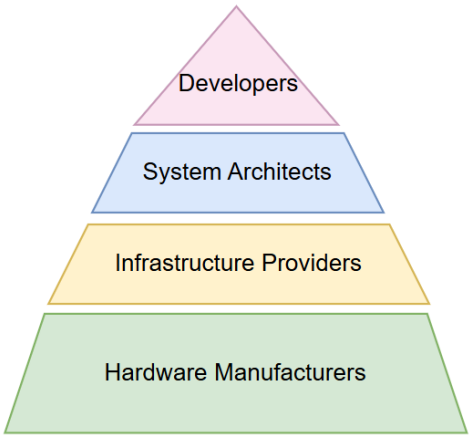


Figure 6.66: AI Inference Stakeholders Organized by System Layer

By adopting a role-based perspective, each group can focus on actionable practices within its domain that, when considered collectively, help improve resource utilisation, contain operational costs, and strengthen the scalability of inference systems across the full AI stack. The recommendations aim not only to guide technical decisions but also to align optimisations across the ecosystem, ensuring that performance gains achieved at one layer do not become inefficiencies at another.

The role-aware recommendations are detailed below:

1. Recommendations for Developers

From the perspective of developers, one of the most critical design choices when building AI systems is the selection of the model to deploy. Based on the benchmarks conducted in this project, a key recommendation is to evaluate candidate models not only by their parameter count, but also by the architectural design of their Attention Blocks, as this aspect plays a central role in determining memory utilisation during inference.

The experiments show that the attention mechanism—particularly its query pattern—has a direct and significant impact on the memory footprint, which in turn shapes the scalability and efficiency of inference workloads. In practice, a higher parameter count does not necessarily imply higher memory consumption. For instance, the Falcon-40B model demonstrated more favourable memory behaviour under load than Qwen-14B, despite having nearly three times as many parameters. This efficiency stems from Falcon's use of multi-query attention, a design

that substantially reduces the memory allocated per token during inference and thereby delays the onset of saturation.

Therefore, developers should incorporate memory scaling characteristics into their model selection criteria, particularly when targeting high-throughput, resource-constrained environments. Choosing models with efficient attention architectures can yield substantial gains in inference performance without requiring additional hardware investment.

2. Recommendations for System Architects

Beyond the selection of individual models or inference frameworks, system architects must make high-level design decisions about how inference workloads are physically and logically distributed across the infrastructure.

A key recommendation derived from the benchmarking results is to avoid building large, multi-purpose GPU clusters that attempt to serve all types of generative workloads equally. Instead, inference infrastructure should be specialised: architects should provision multiple subsystems that are tailored to the computational and memory demands of each application domain.

For example, models serving vision tasks with long image captions or multimodal embeddings behave very differently from chat-oriented models with shorter prompt–response patterns. The concurrency limits, memory saturation thresholds, and latency profiles vary significantly between domains, which directly affects other choices such as sequence length tolerances or number of tensor parallelism utilised among others. Mixing these workloads within the same cluster leads to performance variability and inefficient resource utilisation.

Therefore, a more effective architectural strategy is to deploy multiple homogeneous clusters, each configured for a specific workload family, and route traffic based on those characteristics. This design enables more predictable saturation behaviour, and therefore more efficient horizontal scaling.

Adopting this specialised multi-cluster strategy does involve a clear tradeoff. On one hand, it introduces higher provisioning complexity and may increase short-term infrastructure costs due to dedicated hardware allocation. On the other hand, it enables more predictable saturation behaviour, higher throughput, and more efficient resource utilisation, particularly under sustained or high-volume traffic. As such, system architects must make this tradeoff based on the stringency of the service level objectives (SLOs) they are expected to meet.

3. Recommendations for Infrastructure Providers

Infrastructure providers play a foundational role in the efficiency of large-scale AI systems. Although they do not design models or select inference strategies, they are ultimately responsible for building, maintaining, and offering the physical infrastructure upon which inference workloads run. This includes everything from selecting the hardware to configuring cluster layouts, setting provisioning policies, and exposing infrastructure options to higher-level users such as system architects and platform engineers. As such, their decisions shape the operational envelope in which all other layers of the AI stack must function.

One actionable recommendation is that infrastructure providers, in close collaboration with system architects, should ensure that their service portfolio offers not only general-purpose clusters but also configurations purpose-built for specific workload types. Rather than deploying a single large multi-purpose system, providers should expose a set of specialized, single-purpose infrastructure profiles that align with the diverse nature of AI applications, such as those focused on image processing, text generation, or low-latency streaming. These specialized setups should be readily consumable and easily selectable by the clients who make architectural decisions, enabling better alignment between workload requirements and hardware capabilities.

Because the growing number of configuration choices may introduce complexity and uncertainty for consumers, infrastructure providers are also in a unique position to offer advisory and consulting services that help system architects navigate this landscape. These services could include workload profiling, trade-off analysis, and tailored deployment recommendations, ensuring that clients can extract the maximum value from the infrastructure and make informed choices based on their performance, latency, and cost constraints.

Beyond this, as an additional recommendation, infrastructure providers must carefully weigh the trade-off between resource overprovisioning and Service Level Agreements(SLAs) compliance, in order to keep the system operating at its technical optimum while ensuring that the business remains economically sustainable. To help identify where this ideal operating point resides, a detailed economic study has been developed in Section 4.5, breaking down how the provider's profit structure evolves as a function of the aggregate throughput delivered by the system.

Once this optimal operating region is determined, infrastructure providers must monitor saturation thresholds and user concurrency levels in real time, and implement mechanisms for automated horizontal scaling to keep the system near this ideal operational point. Doing so minimizes the risk of drifting into low-efficiency or negative-profit regions, and helps sustain both high performance and

predictable margins under varying traffic conditions, as discussed in the economic analysis.

4. Recommendations for Hardware Manufacturers

While software and system-level changes can improve efficiency to some extent, the underlying hardware imposes hard limits, and manufacturers are in a unique position to alleviate the root causes of bottlenecks observed during inference at scale.

The multivariable analysis conducted in this project consistently pointed to GPU memory utilisation, with a particular emphasis on the saturation caused by KV-cache storage, as the most prominent bottleneck limiting inference throughput under increasing load. Across all benchmarks, it was memory pressure, not compute saturation, that most often triggered the collapse of throughput and the rise in latency that users experienced.

This insight reveals a critical design priority for the next generation of AI hardware. To enable more efficient scaling of LLM inference, manufacturers should prioritise innovation in memory design, including how memory is sized, structured, and accessed across devices.

Several promising avenues include:

- **Memory expansion through cross-device pooling:** Increase effective memory capacity by aggregating memory across devices using memory pooling techniques. This allows key-value cache entries to overflow gracefully into shared memory spaces rather than causing hard capacity limits on a single device. The recent *Mooncake* proposal exemplifies this approach by introducing CXL-based pooled memory architectures to enable large-scale KV-cache storage beyond local GPU boundaries [59].
- **Disaggregated handling of prefill and decode phases:** Separate the compute and memory allocation strategies for the prefill and decode stages of LLM inference. Prefill stages tend to be short and memory-heavy, while decode stages require longer residency and are more latency-sensitive, so by disaggregating them (as seen in NVIDIA's newly released *Dynamo* project) systems can schedule these phases on heterogeneous hardware more efficiently, reducing peak memory demand and improving GPU reuse [63].
- **Hierarchical KV-cache management across memory tiers:** Design GPU systems to manage KV-cache using multiple memory tiers, such as high-bandwidth GPU memory, slower CPU RAM, and optionally SSD-based storage. Recent work in disaggregated KV-caching proposes promoting and

evicting cache entries between these tiers dynamically, similar to how multi-level CPU caches operate [59]. This extends effective cache capacity and delays saturation, especially when most of the context window is not needed for active decoding.

Chapter 7

Conclusions

This chapter closes the project by revisiting the previously defined objectives and assessing the extent to which they were achieved. Finally, it outlines possible future work, suggesting ways in which the system could be extended or adapted to address new needs.

7.0.1 Achieved Objectives

The primary objectives of this Final Degree Project have been fully achieved. As outlined in Section 4.2, this work set out to investigate the performance and cost-efficiency of LLM inference at scale through a multivariable lens, supported by a robust experimental pipeline and culminating in a set of role-specific recommendations for improving infrastructure-level efficiency. The contributions made across these three fronts are summarised below:

- **Multivariable analysis of LLM inference performance:** Through an extensive suite of experiments varying key parameters such as concurrency, model size, input length, and tensor parallelism, this project systematically revealed how different factors interact to affect inference throughput, latency, and GPU utilisation. The analysis consistently identified GPU memory saturation (particularly coming from KV-cache accumulation) as the primary bottleneck limiting system performance under load. This insight allowed for a deeper understanding of when and why inference systems fail to scale, and how this impacts the cost structure for infrastructure providers.
- **Design and implementation of a reproducible benchmarking pipeline:** A modular, automated benchmarking system was developed to orchestrate model deployment, request generation, and telemetry capture across an H100

GPU cluster. The pipeline ensured repeatability by varying one experimental variable at a time while holding others fixed, capturing telemetry at subsecond granularity, and storing structured outputs for later analysis. This tool enabled the multivariable study to proceed in a systematic and scalable fashion.

- **Empirical foundation for role-aware operating guidance:** The benchmark results were distilled into targeted recommendations for each of the four key roles involved in the AI infrastructure stack: developers, system architects, infrastructure providers, and hardware manufacturers. Each recommendation reflects the practical implications of the performance patterns uncovered during experimentation and offers actionable strategies tailored to the decisions and levers available to each stakeholder. This guidance is intended to help each role operate their portion of the stack more efficiently, thereby improving overall system performance and economic viability.

Beyond fulfilling its original goals, the project delivered a flexible benchmarking system that can be extended to evaluate new workloads, models, and hardware configurations. The reproducibility and clarity achieved in the data collection and analysis process ensure that this work can serve as a foundation for future research and performance tuning efforts in production settings.

7.0.2 Future Work

While this thesis has focused on intra-node benchmarking, the implementation of an automated and reproducible benchmarking system opens the door to broader explorations. The most natural next step would be to extend these experiments to an inter-node setting, where multiple H100 nodes operate together as a full cluster. Such a study was beyond the scope of this work, as the cost of deploying and maintaining multi-node infrastructure can reach hundreds of thousands of dollars.

Investigating this scenario would provide valuable insights into the scalability limits of large language model inference. In particular, it would make it possible to determine under which conditions pipeline parallelism continues to provide performance gains over single-node deployments, and how the choice of model architecture influences these outcomes.

Pursuing this line of work would complete the picture initiated in this thesis, enabling a more comprehensive understanding of inference performance across both single-node and distributed multi-node environments.

Bibliography

[1] Emma Roth. *ChatGPT's weekly users have doubled in less than a year*. https://www.theverge.com/2024/8/29/24231685/openai-chatgpt-200-million-weekly-users. The Verge, 29 Aug 2024. Aug. 2024. (Visited on 08/22/2025).

[2] Kenrick Cai. *Alphabet reaffirms $75 billion spending plan in 2025 despite tariff turmoil*. https://www.reuters.com/technology/alphabet-ceo-reaffirms-planned-75-billion-capital-spending-2025-2025-04-09/. Reuters, 10 Apr 2025. Apr. 2025. (Visited on 08/22/2025).

[3] Reuters. *Microsoft to spend record $30 billion this quarter as AI investments pay off*. https://www.reuters.com/business/microsoft-spend-record-30-billion-this-quarter-ai-investments-pay-off-2025-07-30/. Reuters, 30 Jul 2025. July 2025. (Visited on 08/22/2025).

[4] Wayne Williams. *Nvidia's fastest AI chip ever could cost a rather reasonable $40,000 — but chances that you will actually be able to buy one on its own are very, very low and for a good reason*. https://www.techradar.com/pro/nvidias-fastest-ai-chip-ever-could-cost-a-rather-reasonable-dollar40000-but-chances-that-you-will-actually-be-able-to-buy-one-on-its-own-are-very-very-low-and-for-a-good-reason. TechRadar Pro, 26 Mar 2024. Mar. 2024. (Visited on 08/22/2025).

[5] Reuters. *OpenAI hits $12 billion in annualized revenue, The Information reports*. https://www.reuters.com/business/openai-hits-12-billion-annualized-revenue-information-reports-2025-07-31/. Reuters, 31 Jul 2025. July 2025. (Visited on 08/22/2025).

[6] Yiheng Liu et al. *Understanding LLMs: A Comprehensive Overview from Training to Inference*. 2024. arXiv: 2401.02038 [cs.CL]. URL: https://arxiv.org/abs/2401.02038.

[7] Fanlong Zeng et al. "Distributed training of large language models: A survey". In: *Natural Language Processing Journal* 12 (2025), p. 100174. ISSN: 2949-7191. DOI: https://doi.org/10.1016/j.nlp.2025.100174. URL: https://www.sciencedirect.com/science/article/pii/S2949719125000500.

[8] Yash Sharma. *Decoder-Only Transformers Explained: The Engine Behind LLMs*. https://medium.com/@yash9439/decoder-only-transformers-explained-the-engine-behind-llms-3a3224086afe. Accessed: 2025-08-21. 2023.

[9] Hendrik Strobelt. *Neural Network Embeddings Explained*. https://medium.com/data-science/neural-network-embeddings-explained-4d028e6f0526. Accessed: 2025-08-21. 2019.

[10] Jürgen Schmidhuber. "Deep learning in neural networks: An overview". In: *Neural Networks* 61 (Jan. 2015), pp. 85–117. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2014.09.003. URL: http://dx.doi.org/10.1016/j.neunet.2014.09.003.

[11] Ayon Ghosh. *Cross Entropy in Large Language Models (LLMs)*. https://medium.com/ai-assimilating-intelligence/cross-entropy-in-large-language-models-llms-4f1c842b5fca. Accessed: 2025-08-21. 2024.

[12] Matthew Stewart. *Backpropagation: Step by Step Derivation*. https://towardsdatascience.com/backpropagation-step-by-step-derivation-99ac8fbdcc28/. Accessed: 2025-08-21. 2019.

[13] Jie Gui et al. *A Survey on Self-supervised Learning: Algorithms, Applications, and Future Trends*. 2024. arXiv: 2301.05712 [cs.LG]. URL: https://arxiv.org/abs/2301.05712.

[14] Amit Kumar. *Masked Language Model: All You Need to Know*. https://medium.com/@amit25173/masked-language-model-all-you-need-to-know-12ab35319d09. Accessed: 2025-08-21. 2023.

[15] Alonso Silva. *Next Token Prediction*. https://huggingface.co/blog/alonsosilva/nexttokenprediction. Accessed: 2025-08-21. 2023.

[16] Lorna Franklin. *Supervised Fine-Tuning: Customizing LLMs*. https://medium.com/mantisnlp/supervised-fine-tuning-customizing-llms-a2c1edbf22c3. Accessed: 2025-08-21. 2024.

[17] Shuhe Wang et al. *Reinforcement Learning Enhanced LLMs: A Survey*. 2025. arXiv: 2412.10400 [cs.CL]. URL: https://arxiv.org/abs/2412.10400.

[18] Kar Vaitheeswaran. *How Long Will It Take to Train an LLM Model Like GPT-3?* https://karvai.medium.com/how-long-will-it-take-to-train-an-llm-model-like-gpt-3-d48407198077. Accessed: 2025-08-21. 2023.

[19] Alexandru M. Gherghescu et al. "A Look Into Training Large Language Models on Next Generation Datacenters". In: *arXiv preprint arXiv:2407.12819* (2024). URL: https://arxiv.org/abs/2407.12819.

[20] Zixuan Zhou et al. *A Survey on Efficient Inference for Large Language Models*. 2024. arXiv: 2404.14294 [cs.CL]. URL: https://arxiv.org/abs/2404.14294.

[21] Reiner Pope et al. *Efficiently Scaling Transformer Inference*. 2022. arXiv: 2211.05102 [cs.LG]. URL: https://arxiv.org/abs/2211.05102.

[22] —. *How OpenAI Serves Millions of Requests for GPT Models*. Accessed: 2025-08-22. 2025. URL: https://sderay.com/how-openai-serves-millions-of-requests-for-gpt-models/.

[23] Zachary C. Lipton, John Berkowitz, and Charles Elkan. *A Critical Review of Recurrent Neural Networks for Sequence Learning*. 2015. arXiv: 1506.00019 [cs.LG]. URL: https://arxiv.org/abs/1506.00019.

[24] Daksh Patel. *Understanding the Vanishing and Exploding Gradient Problems in RNNs*. https://medium.com/@helloitsdaksh007/understanding-the-vanishing-and-exploding-gradient-problems-in-rnns-7621efd97605. Accessed: 2025-08-22. 2021.

[25] Merve Durna. *NLP with Deep Learning: Neural Networks, RNNs, LSTMs, and GRU*. https://medium.com/@mervebdurna/nlp-with-deep-learning-neural-networks-rnns-lstms-and-gru-3de7289bb4f8. Accessed: 2025-08-22. 2020.

[26] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL]. URL: https://arxiv.org/abs/1409.0473.

[27] Yacine Bouaouni. *From RNNs to Transformers: A Journey through the Evolution of Attention Mechanisms in NLP*. https://medium.com/@yacinebouaouni07/from-rnns-to-transformers-a-journey-through-the-evolution-of-attention-mechanisms-in-nlp-ef937e2c8d05. Accessed: 2025-08-22. 2023.

[28] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[29] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

[30] Neal Reeves et al. *"The Death of Wikipedia?" – Exploring the Impact of ChatGPT on Wikipedia Engagement*. 2024. arXiv: 2405.10205. URL: https://arxiv.org/abs/2405.10205.

[31] Dan Jurafsky. *Transformer Models*. https://web.stanford.edu/~jurafsky/slp3/slides/transformer24aug.pdf. Lecture slides, Stanford University. Accessed: 2025-08-22. 2024.

[32] DataTechNotes. *Tokenization in LLMs: BPE and WordPiece*. https://www.datatechnotes.com/2025/06/tokenization-in-llms-bpe-and-wordpiece.html. Accessed: 2025-08-22. 2025.

[33] Nicholas Roberts. "How Powerful Are Decoder-Only Transformer Neural Models?" In: *arXiv preprint arXiv:2305.17026* (2023). URL: https://arxiv.org/abs/2305.17026.

[34] Jay Alammar. *The Illustrated Transformer*. https://jalammar.github.io/illustrated-transformer/. Accessed: 2025-08-22. 2018.

[35] Shuxiao Chen et al. "A Simple and Effective Positional Encoding for Transformers". In: *arXiv preprint arXiv:2104.08698* (2021). URL: https://arxiv.org/abs/2104.08698.

[36] Dmitriy Guzhov. *Simplifying Transformer Blocks: A Detailed Mathematical Explanation*. https://medium.com/autonomous-agents/simplifying-transformer-blocks-a-detailed-mathematical-explanation-c422d3e3ef8f. Accessed: 2025-08-22. 2023.

[37] Cameron R. Wolfe. *Decoder-Only Transformers: The Workhorse of Modern AI*. https://cameronrwolfe.substack.com/p/decoder-only-transformers-the-workhorse. Accessed: 2025-08-22. 2023.

[38] Aditya Thiruvengadam. *Transformer Architecture: Attention Is All You Need*. https://medium.com/@adityathiruvengadam/transformer-architecture-attention-is-all-you-need-aeccd9f50d09. Accessed: 2025-08-22. n.d.

[39] Aditya Pande. *Understanding LLMs 1: Introduction to Multi Head Attention*. https://medium.com/@aditya.p22/understanding-llms-9e5486ce36a9. Accessed: 2025-08-22. 2025.

[40] Gianluca Guidi et al. "Environmental Burden of United States Data Centers in the Artificial Intelligence Era". In: *arXiv preprint arXiv:2411.09786* (2024). DOI: 10.48550/arXiv.2411.09786. URL: https://arxiv.org/abs/2411.09786.

[41] Ang Li et al. "Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect". In: *arXiv preprint arXiv:1903.04611* (2019). URL: https://arxiv.org/abs/1903.04611.

[42] Meta Engineering. *RoCE Network: Distributed AI Training at Scale*. https://engineering.fb.com/2024/08/05/data-center-engineering/roce-network-distributed-ai-training-at-scale/. Accessed: 2025-08-22. Aug. 2024.

[43] Adithya Gangidi et al. "RDMA over Ethernet for Distributed AI Training at Meta Scale". In: *Proceedings of the ACM SIGCOMM 2024 Conference (SIGCOMM '24)*. Sydney, NSW, Australia: Association for Computing Machinery, 2024, pp. 57–70. DOI: 10.1145/3651890.3672233. URL: https://doi.org/10.1145/3651890.3672233.

[44] Wikipedia contributors. *InfiniBand*. https://en.wikipedia.org/wiki/InfiniBand. Accessed: 2025-08-22. 2025.

[45] NVIDIA. *NVIDIA DGX B200*. https://www.nvidia.com/en-us/data-center/dgx-b200/. Product page. States "up to 15x" inference performance vs DGX H100. Accessed: 2025-08-22. 2025.

[46] NVIDIA. *NVIDIA Blackwell Platform Arrives to Power a New Era of Computing*. https://nvidianews.nvidia.com/news/nvidia-blackwell-platform-arrives-to-power-a-new-era-of-computing. Press release. Confirms GB200/Grace-Blackwell naming and "up to 30x" LLM inference for GB200 NVL72 vs the same number of H100 GPUs. Accessed: 2025-08-22. Mar. 2024.

[47] Seifeur. *How Many GPUs Are Needed to Train GPT-4?* https://seifeur.com/how-many-gpus-are-needed-to-train-gpt-4/. Accessed: 2025-08-22. n.d.

[48] Zak Killian. *Configuring 100K NVIDIA H200/H100 GPUs Usually Takes Years But Musk Did It In 19 Days*. https://hothardware.com/news/musk-colossus-19-days. Accessed: 2025-08-22. 2024.

[49] Data from the Trenches. *Quantization in LLMs: Why does it matter?* Accessed: 2025-08-22. 2024. URL: https://medium.com/data-from-the-trenches/quantization-in-llms-why-does-it-matter-7c32d2513c9e.

[50] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805.

[51] Neptune AI. *Transformers Key-Value Caching: How it Works and Why it Matters*. Accessed: 2025-08-22. 2024. URL: https://neptune.ai/blog/transformers-key-value-caching.

[52] Sailakkshmi Allada. *Understanding the Two Key Stages of LLM Inference: Prefill and Decode*. Accessed: 2025-08-22. 2024. URL: https://medium.com/@sailakkshmiallada/understanding-the-two-key-stages-of-llm-inference-prefill-and-decode-29ec2b468114.

[53] PyTorch. *PyTorch*. https://pytorch.org/. Accessed: 2025-08-22. 2025.

[54] Subhabrata Mukherjee et al. *Orca: Progressive Learning from Complex Explanation Traces of GPT-4*. 2023. arXiv: 2306.02707 [cs.CL]. URL: https://arxiv.org/abs/2306.02707.

[55] Woosuk Kwon et al. *Efficient Memory Management for Large Language Model Serving with PagedAttention*. 2023. arXiv: 2309.06180 [cs.LG]. URL: https://arxiv.org/abs/2309.06180.

[56] Ramya Prabhu et al. *vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention*. 2025. arXiv: 2405.04437 [cs.LG]. URL: https://arxiv.org/abs/2405.04437.

[57] Tri Dao et al. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. 2022. arXiv: 2205.14135 [cs.LG]. URL: https://arxiv.org/abs/2205.14135.

[58] Bin Lin et al. *Infinite-LLM: Efficient LLM Service for Long Context with DistAttention and Distributed KVCache*. 2024. arXiv: 2401.02669 [cs.DC]. URL: https://arxiv.org/abs/2401.02669.

[59] Ruoyu Qin et al. *Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving*. 2024. arXiv: 2407.00079 [cs.DC]. URL: https://arxiv.org/abs/2407.00079.

[60] vLLM Contributors. *vLLM*. https://github.com/vllm-project/vllm. GitHub repository. Accessed: 2025-08-22. 2025.

[61] NVIDIA. *TensorRT-LLM*. https://github.com/NVIDIA/TensorRT-LLM. GitHub repository. Accessed: 2025-08-22. 2025.

[62] SGLang Contributors. *SGLang*. https://github.com/sgl-project/sglang. GitHub repository. Accessed: 2025-08-22. 2025.

[63] NVIDIA. *Introducing NVIDIA Dynamo: A Low-Latency Distributed Inference Framework for Scaling Reasoning AI Models*. https://developer.nvidia.com/blog/introducing-nvidia-dynamo-a-low-latency-distributed-inference-framework-for-scaling-reasoning-ai-models/. Accessed: 2025-08-24. 2024.

Appendix A

Alignment with the Sustainable Development Goals

This chapter outlines how the present Bachelor's Thesis aligns with the Sustainable Development Goals (SDGs).

The Sustainable Development Goals (SDGs) are a collection of 17 global goals set by the United Nations in 2015, intended to be achieved by 2030. These goals represent an urgent call of action by all countries, and aim to tackle three interconnected dimensions: social, economic, and environmental. Each of those dimensions, encompases a set of objectives to ensure prosperity and the well-being of the planet.

Figure A.1: Sustainable Development Goals (SDG)

Analyzing the purpose of each of the seventeen goals, it has been concluded that the three most closely related to the purpose of developing this project are:

SDG9: Industry, Innovation, and Infrastructure

The SDG 9 focuses on promoting sustainable industrialization, innovation, and building resilient infrastructure to foster technological advancement and inclusive economic growth.

In the context of this project, there has been a direct contribution towards achieving this goal. By analyzing the technological infrastructure performance of large-scale artificial intelligence inference systems and researching optimization strategies tailored to specific workloads, the project facilitates more efficient resource utilization, advances innovation in AI technologies, and supports the development of scalable, robust technological infrastructure.

SDG 12: Responsible Consumption and Production

This project also contributes to SDG 12 by analyzing resource utilization, such as computational power and memory usage, in large-scale large language model inference systems. By benchmarking these systems and identifying performance bottlenecks, the project explores ways to enhance resource efficiency, and such improvements can lead to reduced energy consumption and minimized waste of

computational resources, therefore promoting more responsible, sustainable, and environmentally-friendly technological practices.

SDG 13: Climate Action

Artificial intelligence infrastructures, particularly those used for training and inference of large models, carry a significant environmental cost due to their high energy demands. The analysis of hardware performance conducted in this project contributes to efforts aimed at minimizing the waste of computational resources. By promoting more efficient and purposeful use of these resources, the project directly supports the reduction of energy consumption and, consequently, the environmental footprint of large-scale AI systems.