Politecnico di Torino

Politecnico di Torino

Master's Degree in Ingegneria Informatica (Computer Engineering)

Academic Year 2025/2026

Graduation Session March 2026

Adaptive Layer Placement for Pipeline-Parallel LLM Inference at the Edge

Supervisors:

Alessio Sacco

Guido Marchetto

Doriana Monaco

Candidate:

Giandomenico Lacatena

Abstract

The deployment of Large Language Models (LLM) is shifting from centralized cloud environments toward edge-oriented distributed architectures located closer to data sources, driven by the requirements of real-time applications. One possible strategy to fit these models within limited device memory is using Pipeline Parallelism, which partitions the layers across different nodes to achieve concurrency. However, this approach introduces substantial communication overhead–accounting for up to 40% of total execution time–which can significantly degrade end-to-end performance. Choosing an effective deployment configuration, which must consider factors such as GPU characteristics, inter-node communication latency, and model size, can help mitigate this overhead.

Current state-of-the-art approaches focus on maximizing resource utilization but fail to adapt to frequently changing condition. As a result, they struggle in scenarios where nodes introduce communication bottlenecks that outweigh its computational time. As inter-node link latency directly influences end-to-end performance metrics, even one high-latency link can degrade the overall response time and user-perceived latency.

This research fills this gap introducing an agentic LLM-driven decision making framework capable of evaluating the system conditions and selecting the most efficient layer placement for pipeline parallelism. In this framework, the LLM behaves as an autonomous reasoning agent: it is provided with information about the current network state, GPU capabilities, and model configuration, and uses this context to infer the pipeline layout that minimizes overall response time under dynamic conditions.

Our experimental evaluation on a multi-node testbed with varying latency profiles demonstrates that carefully chosen layer placements–and, in many cases, deliberate underutilization of slower nodes–can outperform full-cluster deployments during distributed inference.

Overall, the work demonstrates the value of using an LLM not only as an inference engine but also as a smart orchestrator for distributed AI systems, proving that LLMs themselves can be the solver of their own deployment challenges.

I would like to express my deepest gratitude to my supervisors, for their time, guidance and support throughout this research.
I would also like to thank my family for their constant encouragement and belief in me, which have been my greatest motivation.
Finally, i would like to thank my friends, especially Gabri, Antonio, Silvia and Simona, who shared this journey with me and contributed with encouragement, support, and many memorable moments. The time we spent together helped me to make this path much more enjoyable, and I will never forget it.

Table of Contents

List of Tables

List of Figures

Glossary

AI
: Artificial Intelligence

LLM
: Large Language Model

KV
: Key-Value

IIoT
: Industrial Internet of Things

PP
: Pipeline Parallelism

TP
: Tensor Parallelism

ILP
: Integer Linear Programming

MSE
: Mean Squared Error

DP
: Dynamic Programming

TTFT
: Time To First Token

ITL

Inter-Token Latency

TPOT

Time Per Output Token

RTT

Round-Trip Time

UX

User Experience

GPU

Graphics Processing Unit

Chapter 1

Introduction

Large Language Models (LLMs) such as ChatGPT, LLaMA and Mistral are rapidly advancing the state of natural language processing, allowing a wide range of applications including conversational agents, real-time analytics and autonomous decision-making systems. As these models continue to grow in size and complexity there is an increasing demand to deploy them closer to the data sources, at the edge, in order to satisfy strict latency, privacy and reliability requirements. Traditionally LLM inference is performed using centralized cloud environments, but while powerful, this approach can't meet all the requirements needed. This becomes critical for sectors such as autonomous driving, healthcare and industrial IoT (IIoT), where millisecond delays can impact safety and efficiency.

Edge deployment offers several benefits over centralized cloud computing:

- **Lower latency**, by moving the computation closer to the data sources, eliminating the need to use global networks to transmit data;
- **Improved Privacy Reservation**, since sensitive data such as users information, camera images and sensors data can be processed locally rather than being transmitted externally, this lead to a reduced risk of interceptions on external resources;
- **Reduced Cloud Resources Dependency**, decreasing reliance on always up cloud connectivity and remote infrastructure, which improves system robustness.

However deploying LLMs on edge devices has many constraints regarding GPU memory and heterogeneous hardware, in fact the massive memory footprint, the computation demands and the nature of the transformer architecture make the deployment on resource-finite edge devices infeasible. This has led to an increased interest in **distributed inference strategies** able to span multiple edge devices resources.

Despite their potential, distributed inference methods face several bottlenecks:

- **Computation limitations**: Transformer layers require dense matrix multiplications that often exceed the process capabilities of constrained hardware.
- **Memory overhead**: Intermediate activations and especially the KV (Key Value) cache significantly increase the memory requirements during the autoregressive decoding.
- **Communication latency**: Distributed inference (and training) involves frequent data exchange between nodes. A single slow link can dominate and lead to high latencies.
- **Heterogeneity**: Actual deployments involve nodes with different GPU capabilities, memory capacities and network characteristics, complicating deployment strategies.

To exploit concurrency and fit these large models into these environments Pipeline Parallelism has emerged as a promising solution by **partitioning model layers across multiple nodes**. Communication overhead can account up to 40% of the total execution time, making the choosing a correct pipeline configuration critical, existing approaches generally optimize resource utilization under fixed network conditions and fail to dynamically adapt when those conditions change.

Current state-of-the-art systems neglect the dynamic nature of edge environments, they are designed to maximize throughput under stable conditions, rather than continuously adapt under fluctuating network conditions. As a result they struggle when communication cost outweighs computation, especially when one or more nodes show poorly network conditions and high latencies.

This thesis addresses this gap by introducing a **novel agentic LLM-driven framework** that autonomously selects the most efficient pipeline parallelism configuration based on real-time system conditions. The LLM acts as a reasoning agent: it firstly analyzes the current network latencies, the GPU characteristics and the deployed model architecture, then it suggests the layer placement that minimizes the response time. This enables the system to adjust the pipeline configuration dynamically, sometimes by intentionally avoiding slower nodes when the communication cost outweighs the computation contribute, to achieve lower inference latency.

Experiments conducted on a multi-node testbed with varying latency profiles demonstrate that this adaptive decision-making strategy outperform full-cluster deployment under unstable network conditions, these results show the potential of LLMs not only as powerful inference engines but also as **self-optimizing orchestrators** able to improve their own deployment performance.

1.1 Thesis Structure

The thesis is organized as follows:

- **Chapter 2 - Related Works**: presents prior research on distributed LLM inference, pipeline parallelism, KV-cache optimization in distributed systems, edge to cloud collaborations.
- **Chapter 3 - Background**: Introduces key concepts required to understand the proposed solution of this thesis, including transformers architecture, pipeline parallelism, communication overheads, evaluated metrics.
- **Chapter 4 - Profiling Distributed Inference**: Provides a view of the metrics obtained through profiling a distributed inference system. This chapter visualizes through graphs the communication-to-computation ratio that motivate our thesis work.
- **Chapter 5 - Proposed Solution**: Describes the architecture of the agentic LLM-driven decision making system proposed as solution.
- **Chapter 6 - Experimental Evaluation**: Presents the metrics used for the evaluation, the multi-node testbed exploited for the evaluations, the different latencies scenarios and the evaluation of the obtained results from the benchmarks. This chapter compares also the different LLM reasoning capabilities of the models.
- **Chapter 7 - Conclusions and Future Work**: Summarizes the contributions of the thesis and outlines opportunities for enhancing the framework in the future.

Chapter 2

Related Works

Throughout recent years, multiple research directions have been explored in the study of distributed Large Language Models inference, each offering different mechanism to address computational and communication challenges of running modern AI models out of datacenters. This chapter reviews the literature relevant for our thesis objective, describing modern studies on distributed inference and deployment of Large Language Models (LLMs) and particularly on how they are used on edge devices, for every theme we summarize their work, highlight their strength and identify their limitations that motivate our thesis project approach for every reviewed topic.

The chapter is organized as follows:

- **Distributed LLM inference related works**, this section reviews modern techniques for partitioning and deploying LLMs across multiple devices, including frameworks designed for edge or edge-cloud environments, offering a large view on how they address device heterogeneity, unstable network conditions and limited computation capabilities

- **KV Cache related works**, this section reviews methods that aim to reduce the memory footprint and communication cost of the KV cache during the autoregressive decoding of LLMs;

2.1 Distributed LLM inference related works

Distributed inference is the keystone of running LLMs on resource-limited devices.

This sections provides an overview of the approaches that distribute LLM computation across multiple devices, edge nodes and cloud servers. The analyzed works address together how to partition models, reduce the communication time

between components and maintain high performances despite having hardware heterogeneity and network limitations. We discuss and analyze the following papers.

2.1.1 Distributed Training of Large Language Models

The survey by Zeng et al. [1] provides a comprehensive overview of the main strategies to distribute heavy training workloads across multiple computing nodes, although this work focuses on distribute training but the same parallelization paradigms are used in distribute inference systems, including data parallelism, model parallelism (e.g. pipeline, tensor), 3D parallelism and other modern optimization strategies. As highlighted in the paper the transition from single-GPU to multiple computing nodes relies on several new partitioning strategies. **Pipeline Parallelism** (PP) is particularly relevant for edge nodes because it allows models to be split by layers and place the divided parts on different devices, but while successful at reducing memory load on nodes this method introduces "bubble times" and heavy communication overhead, further optimization like GPipe and PipeDream improved the naive implementation. **Tensor Parallelism** (TP) on the other hand, it's an intra-layer partitioning, where the tensor is divided into N pieces, requiring high-frequency synchronization primitives (All-Reduce/All-Gather operations) which are often unsuitable in low-bandwidth environments.

2.1.2 DILEMMA: Joint LLM Quantization and Distributed LLM Inference Over Edge Computing Systems

To limit the overhead introduced by Pipeline Parallelism, **DILEMMA** [2] introduces a joint optimization approach in order to reduce total inference time, the framework treat quantization and placement simultaneously using Integer Linear Programming (ILP) to decide which layers to place on which edge server and using a certain bit precision. The infrastructure consist of a 2-tier Edge Computing system, combining a set of edge servers and a cloud server (Fig. 2.1).

Point of strength: using Knowledge Distillation it's able to balance the trade-off between inference latency and model accuracy.

Limitations: ILP provides exact solutions but suffers from major practical limitations. It scales poorly with large LLMs and heterogeneous edge environments because the number of decision variables grows exponentially, making solving time and memory usage very high. ILP also requires static, deterministic system conditions and must be fully re-solved whenever workloads or network states change, preventing real-time adaptation. As a result, ILP is often too slow, too rigid, and too computationally heavy for dynamic edge-based LLM inference.

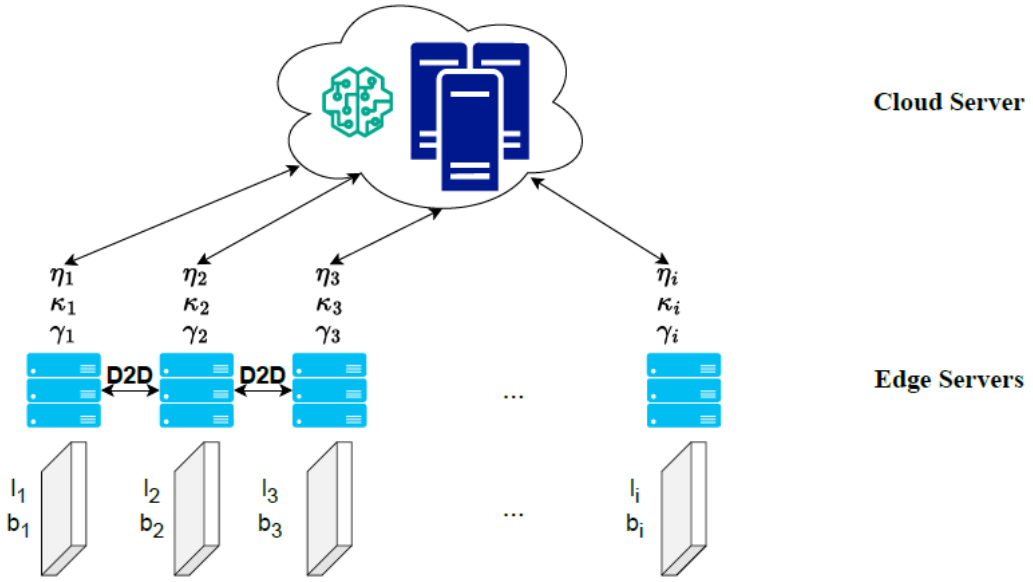

Figure 2.1: Two Layers System Architecture [2].

2.1.3 Distributed On-Device LLM Inference With Over-the-Air Computation

A different approach has been explored by Zhang et al. [3], exploring an **"Over-the-Air" computation method**, who investigate a way to reduce communication overhead in wireless environments. The framework proposed is based on tensor parallelism and in order to mitigate the bottleneck generated from the many all-reduce operations an over-the-air computation method is proposed, this follows the same concept as what in wireless multiple-access channels is called superposition property. By exploiting this wireless channels property they allow devices to perform the sum operation combing signals from multiple devices simultaneously, speeding up the aggregation process.

Point of strength: reduces the communication bottleneck of Tensor Parallelism, reducing latency and maintaining a low Mean Squared Error (MSE).

Limitations: requires special MIMO transceiver hardware and is highly sensitive to the wireless channel state, making this choice less suitable for real less reliable edge networks.

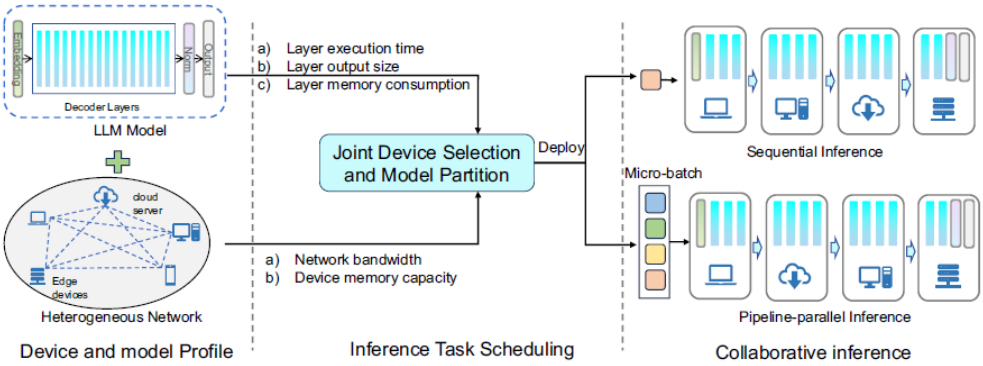

Figure 2.2: Frameworks of EdgeLLM. It consists in three stages: 1) offline profiling, 2) task scheduling optimization; and 3) online collaborative LLM inference[4].

2.1.4 EdgeShard: Efficient LLM Inference via Collaborative Edge Computing

EdgeShard [4] is a framework designed for edge-cloud inference collaboration. The objective is partitioning the models into shards that are manageable for resource-constrained edge devices while using efficiently all the resources of the collaborative environment. The framework addresses device heterogeneity and bandwidth limitations by formulating a joint device selection and model partition problem to optimize inference latency and throughput [4], this is solved through two dynamic programming (DP) algorithms. The framework workflow is shown in figure. 2.2.

Limitations: EdgeShard relies on an offline profiling phase to obtain traces and network metrics, which prevents it from reacting to sudden changes in network latency during runtime, furthermore the framework assumes a centralized scheduler with global visibility on memory and network information of every node in the network, creating a single point of failure and a significant communication overhead.

2.1.5 A Matching Game for LLM Layer Deployment in Heterogeneous Edge Networks

A different approach to distributed inference is explored by Picano et al. [5] who propose a framework based on the **matching game theory** to orchestrate LLM layers across heterogeneous edge nodes assuming a sequential pipeline execution [5]. The system is modeled as a two-sided matching game where LLM layers express their preferences for specific edge nodes based on computational capacity and latency, while nodes prioritize layers looking at resource availability. The algorithm aims to minimize the inference latency by considering computation costs, inter-layer

communication delays and pipeline bubble time. The matching process iteratively assigns layers to nodes until a stable configuration is reached, ensuring that any changes would not benefit from deviating the final allocation.

Point of strength: The framework is decentralized and scalable.

Limitations: The matching game looks at preferences list based on metrics like processing time and transmission time without dynamically checking for network changes during the inference workload.

2.1.6 Discussion and Research Gap

The reviewed works demonstrate really significant progress in distributing model inference across heterogeneous systems, however many limitations remain when we transition from closed controlled environments (e.g. datacenter) to highly dynamic and unreliable edge environments. Current state-of-the-art approaches like the ones we analyzed can be categorized by their reliance on specialized hardware, centralized orchestration or offline profiling and static initial conditions.

- **Hardware dependencies**: Works like Zhang et al. [3] introduce innovative communication primitives (Over-The-Air computation) but requiring specialized MIMO hardware and quite stable wireless conditions, limiting their usability to wireless-only systems without having applicability in standard edge or edge-cloud contexts.

- **Computational complexity**: Frameworks such as DILEMMA [2] and EdgeShard [4] rely on optimization techniques that become increasingly expensive as the model and system scale. ILP and DP limit scalability and making real-time adaptation challenging. As a result, both frameworks face substantial computational overhead when deployed in large or highly dynamic edge systems.

- **Reliance on static conditions and offline profiling:** One of the most critical limitation is the dependency on pre-calculated data. EdgeShard [4] requires an offline profiling phase to obtain execution traces, operating on the assumption that the environment will keep these offline profiles during runtime; the Matching Game [5] approach instead constructs the preference lists once without recalculate them during runtime, consequently they are unable to adapt to sudden network changes occurring out of the scope of the initial profiling data.

Our thesis addresses these challenges by introducing an **agentic LLM-driven framework**. Unlike the discussed works that are bounded by static profiles or fixed preference lists our approach utilizes the LLM as a reasoning agent to perform

real-time interpretation of the environment, observing live conditions rather than relying on offline profiling, the framework can dynamically adjust the pipeline configuration.

2.2 KV Cache related works

In autoregressive LLM inference the **KV cache** is often the primary memory and communication bottleneck. This section investigates the state-of-the-art works on how KV cache can be compressed, shared and optimized in multi devices environments. These works explore some dynamic optimization resource-aware approaches like our thesis but focusing on another important limitation, the KV cache makes possible to avoid redundant computations during the autoregressive computation, it can occupy around 30% of the GPU memory becoming one of the main concern to worry about in distribute systems.

2.2.1 Efficient Memory Management for Large Language Model Serving with PagedAttention (vLLM)

A major advancement in KV cache optimization is the introduction of **PagedAttention** by Kwon et al. [6] which is the base core of the **vLLM** inference serving engine. The framework treats KV cache similarly to virtual memory in traditional operating systems, partitioning the cache into non-contiguous block and eliminating the internal fragmentation, and so increasing the throughput of the served LLM. The evaluation and the experiments of our thesis has been conducted using this state-of-the-art serving engine, while our thesis optimizes the distribution of the workload on nodes based on network conditions, vLLM optimizes the node's memory reserved to the KV cache obtaining the maximum performance.

2.2.2 Compression and Quantized Computation works

Recent researches have focused on reducing the size of the cache before the transmission. For example **CacheGen** [7] encodes the KV cache into bitstream, similarly to what happens in video encoding, to reduce network transfer size, using Delta Encoding and Arithmetic Coding as encoding operations. Existing quantization method like CacheGen compresses KV but requires dequantization at every step, this leads to high computation overhead, on this purpose **HACK** [8] has been introduced, allowing matrix multiplications directly on quantized INT2/INT8 KV data by using homomorphic quantization. Using this technique HACK reduces Job Completion Time (JCT) by up to 70.9% over disaggregated baseline and up to 52.3% JCT reduction over quantization methods (CacheGen). The HACK overview is illustrated in figure 2.3.

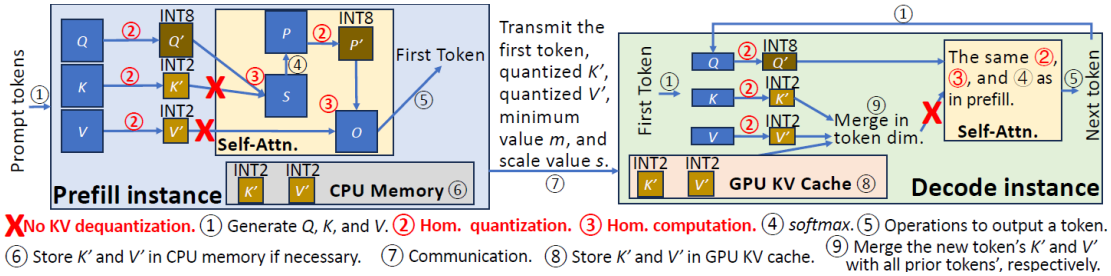

Figure 2.3: Overview of HACK for disaggregated LLM inference. Taken from [8]

DroidSpeak [9] introduces the concept of sharing KV caches between different models, by identifying "critical layers" (the 10% most sensitive to KV Cache) it recomputes only essential data. For example in Multi-agent LLM different fine-tuned models sharing the same input are used, this leads to calculation redundancy and slower inference time especially in the prefill phase (first token generation), where the full input is analyzed before the output generation.

Chapter 3

Background

This chapter introduces some theoretical foundation necessary to understand the mechanism of Large Language Models and the proposed agentic LLM-driven framework of this thesis.

First, we briefly introduce the basics of machine learning and neural networks. Then we introduce Large Language Models (LLMs) and their Transformer-based architecture. Next, we describe the cloud and edge computing paradigm, explaining how modern AI systems are deployed. Finally we discuss the distributed inference techniques that enable the execution of LLMs across heterogeneous devices focusing on pipeline parallelism, we also explain the evaluations metrics used in this thesis.

3.1 Machine Learning

Machine Learning (ML) is a subfield of study in artificial intelligence that focuses on the development of algorithms capable to learn patterns from data and to make predictions and decisions without being explicitly programmed. Instead of relying on manually assigned rules, ML systems automatically learn the relations between inputs and outputs by analyzing data during a training process.

ML methods are usually divided into three categories:

- **Supervised learning**, the model is trained using labeled examples;
- **Unsupervised learning**, the model discovers patterns using unlabeled data;
- **Reinforcement learning**, an agent learns to make actions through interactions with its environment. The agent can choose between make exploration or exploitation.

During recent years, the most successful ML systems have been based on **Deep**

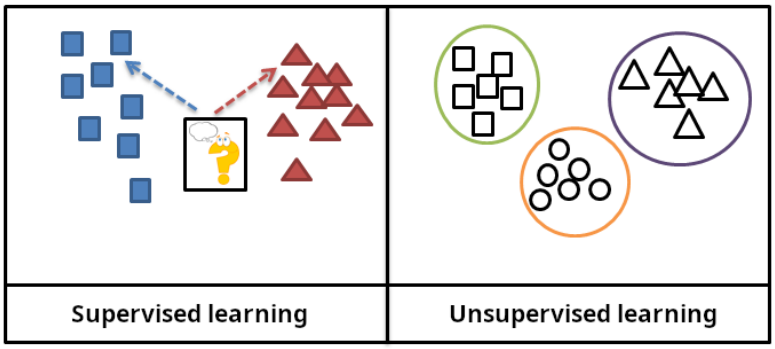

Figure 3.1: Difference between supervised and unsupervised learning. [10]

Learning, a subdiscipline of ML, focusing on models composed of multiple layers of neural networks capable of learning hierarchical features of data.

3.1.1 Neural Networks

Artificial neural networks are computational models inspired by the structure of the human brain. They consist of layers of interconnected nodes called neurons, each performing a weighted sum of its inputs followed by a nonlinear activation function to produce the output.

Many stacked layers gives rise to what is known as **Deep Neural Networks**, those are able to learn complex relations and have become the foundation of vision and speech recognition, together with natural language processing.

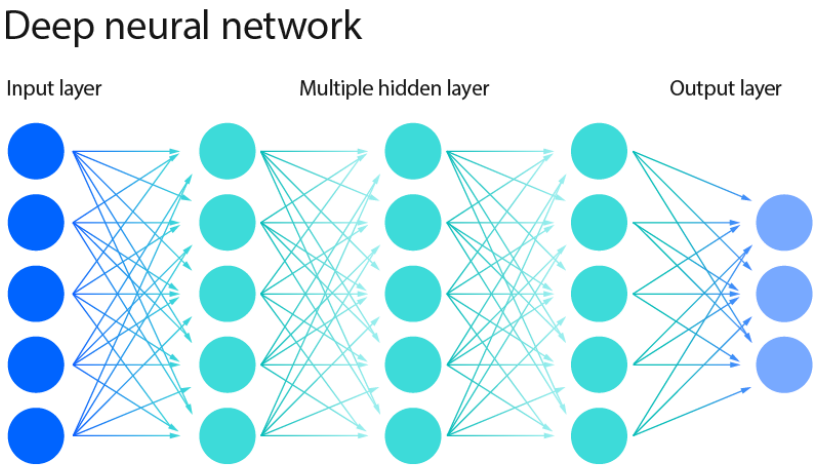

Figure 3.2: Deep neural network illustration. [11]

3.2 Large Language Models

Large Language Models are deep neural networks trained on massive text-based dataset to model natural language. Their objective is to predict the probability of the next token given a sequence of previous tokens. LLMs were firstly based on recurrent neural networks but they are now based on the **Transformer** architecture, introduced by Vawasani et al. in 2017 [12]. Transformer relies on a mechanism called **self-attention**, this allows the model to capture long-range dependencies between sequence of tokens.

Compared to previous recurrent architecture they enable more efficient parallelization and have demonstrate better performances on a multitude of tasks related to natural language processing, such as translation, summarization and typical questions answering.

3.2.1 Transformer Architecture

The Transformer architecture is composed of a stack of identical layers (see Fig. 3.3), each containing two core components:

- **Multi-Head Self Attention**: allows to understand the context of a token by simultaneously looking to multiple relationships within a sequence;
- **Feed-forward neural networks**: once the attention mechanism acquired all the contexts needed from other tokens it transforms the current token representation processing the gathered information.

Self-Attention computes interactions between all the tokens in the sequence through three representations known as queries (Q), keys (K) and values (V). This mechanism enables the model to focus on the most relevant context when processing each token.

In modern LLMs the large number of parameters allow the capture of complex linguistic structures but the tradeoff is requiring high computation and memory during inference.

3.2.2 Autoregressive decoding and KV cache

Autoregressive generation in LLMs refers to the technique of producing new data by predicting the next element in a sequence based on all preceding elements. To make this process efficient previous computed attentions are stored in a **KV cache**, holding the key (K) and value (V) tensors. For each new token the K and V entries are simply appended, avoiding the need to recompute the attention over all past tokens every time. While this reduces computation it increases the occupied

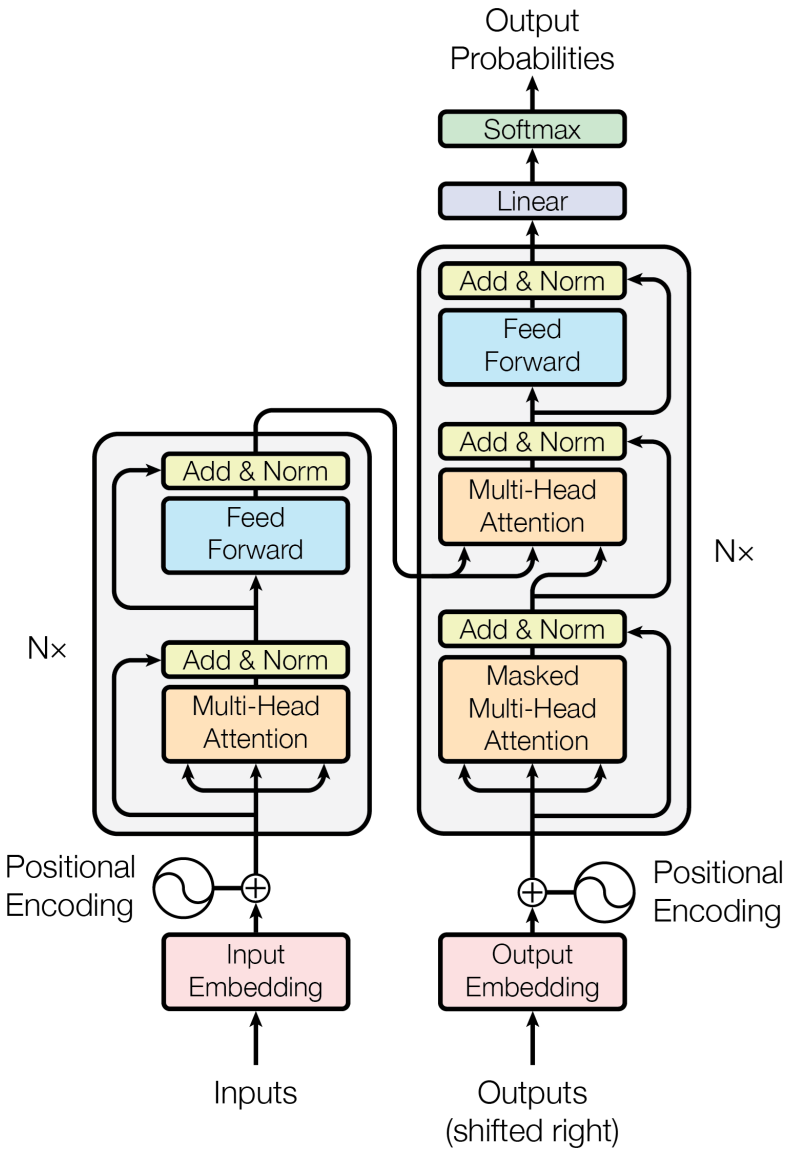

Figure 3.3: Picture taken from [12]

memory, in distributed environments it also increases communication volume when the KV must be shared across different entities.

3.3 Edge and Cloud computing

Modern AI systems are nowadays deployed across distributed infrastructures that include both centralized cloud datacenters and decentralized edge devices.

Cloud computing provides high computational resources such as GPUs and high-performance interconnections, allowing intensive computation tasks such as training and serving of large AI models. However cloud-only deployments may introduce significant latencies, large data transfers and privacy concerns. To address these limitations, the concept of **edge computing** has emerged. Edge computing moves part or all the computation closer to the data sources, edge devices can comprehend IoT devices, vehicles, smart cameras, vehicles or local servers.

Edge computing offers several advantages:

- Reduced latency for time-aware applications
- Lower network bandwidth usage
- Improved privacy, by keeping data closer to the source

Although edge devices are usually constrained in terms of computational capacity, memory size and energy consumptions. These limitations make the deployment of large models particularly challenging.

3.3.1 Edge–Cloud collaboration

Hybrid approaches place latency-sensitive parts at the edge and heavier tasks in the cloud, sometimes using the cloud just as data backup. These approaches require a careful partitioning of the computation between resources but can combine the strength of both type of environments.

3.4 Distributed Inference

Distributed inference refers to the serving of a machine learning across multiple computing devices instead of a single node. This approach allows large models to be executed even when the individual device lacks sufficient memory or computation resources.

There are many different strategies that can be used to distribute model execution.

3.4.1 Data Parallelism

In data parallelism, multiple replicas of the same model are deployed on more devices, each device processes a different portion of the input data. This approach is mainly used during model training but is also useful when there is a need to optimize the computational workload in a distribute inference setup. However, Data Parallelism doesn't reduce the memory footprint of the model since every device must have a full copy of the model parameters.

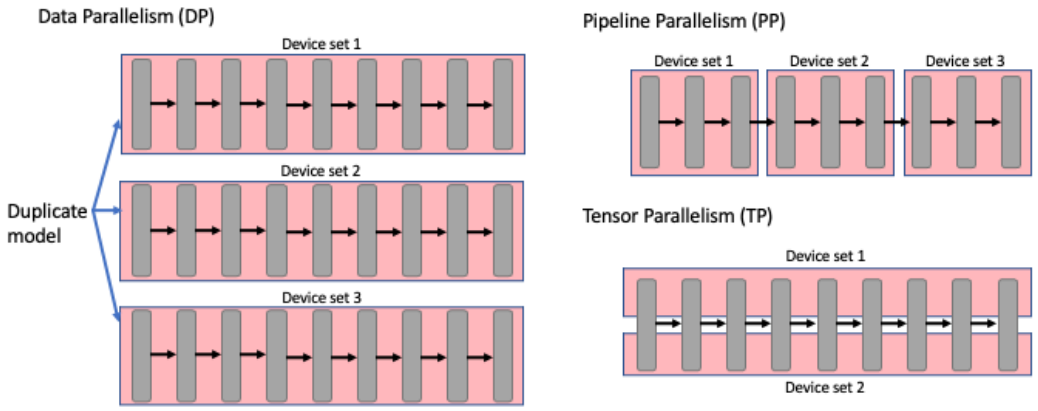

Figure 3.4: Different parallelism illustration. [13]

3.4.2 Model Parallelism

Model parallelism divides the neural network across multiple devices by partitioning its parameters or the computation. This is particularly helpful for large models that are not able to fit into the memory of a single device.

The most common parallelism techniques are:

- **Pipeline Parallelism**, the model is divided into group of layers, each group is assigned to different devices. During the computation the inputs are processed sequentially across the pipeline. This approach reduces the occupied memory on devices but introduces pipeline bubbles and communication overhead between ranks.

- **Tensor Parallelism**, instead of dividing the model per layers, this parallelism splits each tensors within a layer across different devices. This method requires frequent synchronization operations such as all-reduce and all-gather between devices to sync intermediate activations,

- **Expert Parallelism**, normally used in Mixture-of-Experts (MoE) architectures, expert parallelism allows the distribution of different expert across devices. During inference only a bunch of experts is active for each token, allowing the model to have a big number or parameters but keeping the computational cost low.

For distributed inference systems one of the most suitable is Pipeline Parallelism, it allows each device to execute a different portion of the model, allowing models larger than the memory capacity of individual nodes to be deployed smoothly.

However distributed inference leads to additional challenges between the devices in the network, including communication overhead and high sensitivity to network latency. In heterogeneous edge environments where network conditions are often unstable these issue become very relevant.

Distributing LLM computation across such environments is a significant research problem, and this thesis focuses on designing efficient strategies to address it.

Chapter 4

Profiling Distributed Inference

This chapter presents an analysis of Large Language Models (LLM) inference across distributed nodes. By **profiling** the two standard parallelism strategies, Tensor Parallelism (TP) and Pipeline Parallelism (PP), we identify the performances and the the possible bottlenecks that can emerge in heterogeneous and unreliable network environments. These findings provide some motivations for the dynamic orchestrator framework introduced in the next chapter.

4.1 Experimental setup

To make sure the results reflect typical transformer-based architectures we utilized the OPT (Open Pre-trained Transformer) Language Models family by Meta AI [14], ranging from 125M to 6.7B parameters (see Table 4.1). All the experiments were conducted on the the Fabric Testbed [15] using NVIDIA A30 GPUs, leveraging vLLM to run these models in inference mode, processing 400 prompts together to achieve some significant statistical results, the job has been run multiple times to get some accurate confidence intervals.

Table 4.1: Architectural specifications OPT Meta models

Model	Hidden size	Heads	Layers	Parameters
OPT-125M	768	12	12	125M
OPT-350M	1024	16	24	350M
OPT-1.3B	2048	32	24	1.3B
OPT-6.7B	4096	32	32	6.7B

4.2 Initialization time and Throughput

One of the critical phase in distributed inference is the KV Cache initialization, having a smaller time spent for this can significantly impact the overall responsiveness of the system. As shown in Figure 4.1, Pipeline Parallelism (PP = 3) even it has more ranks reduces the initialization time by approximately **50%** compared to Tensor Parallelism (TP = 2) for the OPT-6.7b model. This is related to the simpler mechanism to achieve layer distribution, in fact the KV-Cache is distributed in the following way:

- **TP**: The KV Cache is distributed across GPUs, each GPU contains a portion of key and values matrices, every GPU stores only the values corresponding to the attention heads it manages (e.g. with 2 GPUs, each GPU holds half attention heads). GPUs don't exchange the entire KV Cache but they operate on their local shard When some layers need to be rebuilt communication occurs through AllReduce and AllGather primitives.
- **PP**: The KV Cache is stored for each GPU only for the layers assigned to it; as a result they have different KV Caches while they exchange only the necessary intermediate activations through Send and Recv primitives.

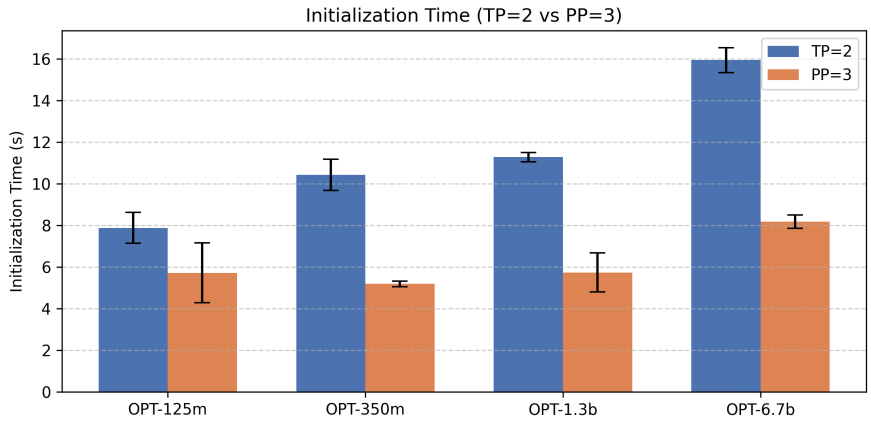

Figure 4.1: KV cache initialization time for different model size and two distributed configurations: TP within 2 GPUs, PP within 3 GPUs.

Although, if we examine the throughput achieved in Figure 4.2 in terms of tokens/s we can see some different results. When using TP, the throughput is much higher compared to PP despite using less GPUs because it allows the devices to work on different parts of the same layer simultaneously, there is less synchronization overhead and less "bubble time" [1], resulting in higher GPU usage and faster token

generation. However, this difference is lower for bigger models, as more data is exchanged through collective operations which reduces the benefit got from the parallelization. PP shows a consistent performance across different model sizes, as it divides the model layers across GPUs and the throughput is limited by the pipeline stages.

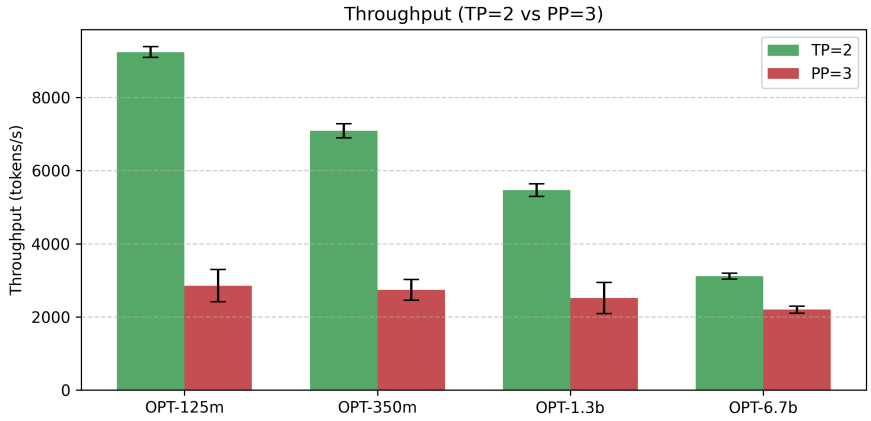

Figure 4.2: Throughput achieved by TP within 2 GPUs and PP within 3 GPUs with different model sizes.

4.3 Communication–Computation Ratio

The most significant finding in our profiling research is the ratio between communication and computation. Figure 4.3 shows the percentages of the total communication time spent for tensor parallelism (TP=2) and pipeline parallelism (PP=3) across different model sizes. PP shows a considerable higher communication overhead compared to TP, and it become larger as the model size increases. Instead a more stable communication overhead is shown in TP, where the percentage remains nearly constant across different sizes. Looking at Figure 4.3, with PP we can observe that the communication overhead can reach up to **40%** of the total execution time for larger models such as OPT-6.7B. This behavior highlight a key difference between the two parallelism. PP simplifies the layer distribution across devices but requires more inter-layer dependencies during computation. In fact each stage must wait to receive the activations from the previous stage to start computing, as shown also in Fig. 4.4. TP, on the other hand, requires more synchronization within every layer but it achieves a better ratio between communication and computation.

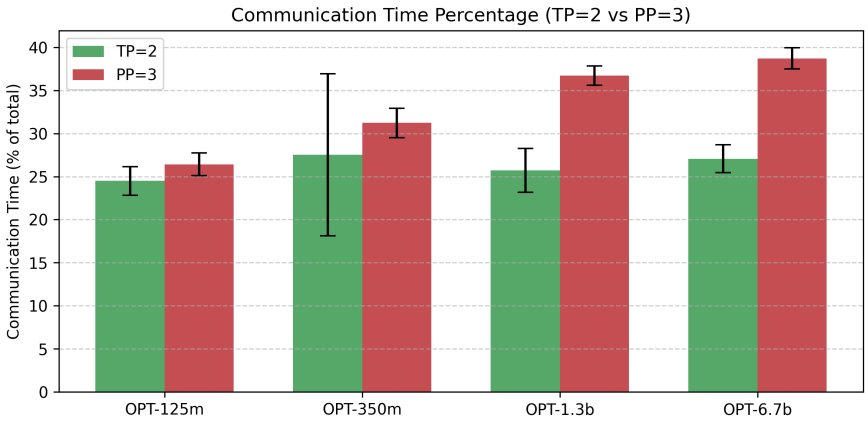

Figure 4.3: Communication time percentage compared to total time achieved by TP within 2 GPUs and PP within 3 GPUs with different model sizes

4.4 Timeline Analysis Comparative

To visually understand how these parallelism work, we analyzed the timelines for both strategies using the OPT-6.7b model:

4.4.1 Pipeline Parallelism Timeline

The Pipeline Parallelism timeline in Figure 4.4 reveals large idle gaps (bubbles), one inefficiency introduced by this strategy, during these gaps the GPUs are waiting the datas from other stages. What we can actually see: Rank0 only have to send the activations onto the next stage and Rank2 has to receive the activations from Rank1, consequently Rank1 must send and receive activations from adjacent ranks. In an edge environment where latencies are high, these bubbles are bigger, causing a slowdown in the overall performances and output token generations.

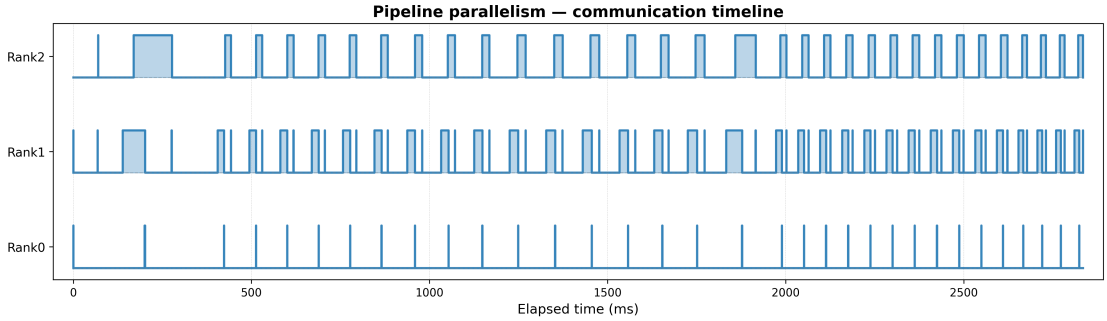

Figure 4.4: Communication activity trace for Pipeline Parallelism (PP). The vertical columns represent send/recv durations, while the gaps between them highlight the "Pipeline Bubbles" where the system stalls due to data dependencies

4.4.2 Tensor Parallelism Timeline

In contrast, the Tensor Parallelism timeline in Figure 4.5 shows thousands of micro-bursts (All-Reduce and All-Gather operations). While there are fewer "idle" periods, the high frequency of synchronization makes TP fragile especially in edge environments. A moderate delay on a link might cause the entire model to stall thousands times per seconds.

4.5 Considerations for Dynamic Placement

Our profiling proves that distributed LLM inference cannot be addressed with a single static configuration. Static placement are most likely inefficient, for example

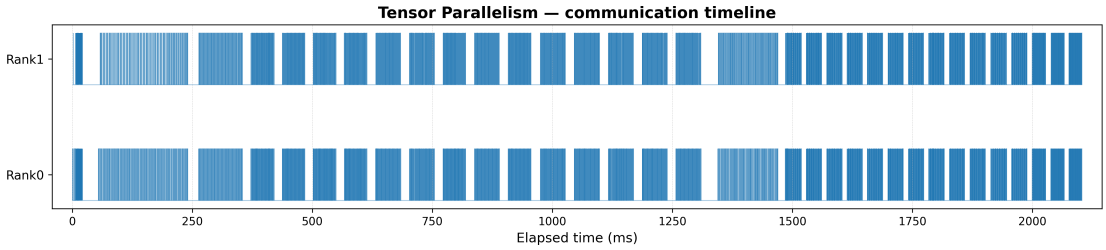

Figure 4.5: Communication activity trace for Tensor Parallelism (TP). The dense micro bursts illustrates the high frequency of synchronization required, demonstrating also why TP is more sensitive to network latency compared to PP.

a fixed $8 - 8 - 8 - 8$ partitioning is optimal if all links are fast and stable, there is also a clear opportunity to improve performances by intentionally avoiding slower nodes for the distributed inference to shrink the "bubbles" described in Section 4.4.1. These observations motivate our thesis work, if a system can reason about the latency conditions it can repartition the model layers to improve the communication timeline, even under fluctuating network conditions. These put the basis for the Agentic LLM-driven Orchestrator introduced in the next chapter.

Chapter 5

Solution

This chapter presents the proposed framework designed to improve the deployment of Large Language Models in distributed edge environments. As discussed in the previous chapters, running LLM across heterogeneous devices introduces several challenges, including limited computation and memory resources, heterogeneous hardware and fluctuating network conditions that can affect communication latencies.

Traditional distributed inference systems usually rely on static placement strategies or optimization algorithms based on offline metrics. This is effective in controlled environments but these approaches frequently struggle to adapt to changing network conditions usually observed in real edge environments.

To address these limitations this thesis proposes an **agentic LLM-driven framework** that leverages the reasoning capabilities of modern Large Language Models to dynamically determine the most efficient pipeline configuration. The system continuously evaluates the current state of the devices in the collaborative network, selecting the pipeline assignment that minimizes the latency and maintaining a balanced performance across evaluation metrics.

This proposed solution introduces an **agentic decision-making framework**, implemented using an LLM, this acts an orchestrator capable of reasoning on the system conditions and recommending the most appropriate layer placement across all the available nodes, when the communication cost outweighs the computation contribute on a node, and the model deployed can still maintain good performances, it can intentionally exclude the node from the configuration to achieve better inference latency. This method avoid configurations that would lead to communication bottlenecks.

The rest of this chapter describes the implementation of the proposed framework. The problem is presented and is followed by an overview of the distributed system architecture. Next, the prompt to determine the correct pipeline configuration is explained along with the mechanism used to collect network metrics and execute

the chosen configuration strategy.

5.1 Problem Definition

Deploying Large Language Models using Pipeline Parallelism across multiple nodes requires to determine how models layers should be distributed among the available devices. The main goal of the pipeline partitioning selection is to minimize the overall inference latency while taking into consideration hardware constraints and communication costs between the nodes. In heterogeneous edge environments the optimal pipeline configuration can vary over time due to network latency fluctuations or nodes availability. Some configurations that work well under certain conditions may become not efficient when communication delays increase.

We can summarize the problem addressed by this thesis as the dynamic and continuous selection of a pipeline configuration that aims to optimizes performance given the current conditions.

5.2 System Architecture Overview

The proposed framework in this thesis is built on a 2 layers architecture, it decouples the **orchestration** from the **execution** by introducing an **Agentic LLM-driven orchestrator** and a **high-performance Distributed Inference Runtime**. This separation ensures that the overhead introduced by the decision-making task doesn't interfere with the distributed inference executions and impacting their speed. While the inference engine focuses on achieving fast inference using vLLM, the agentic orchestrator exploits an LLM-driven reasoning agent to dynamically configure the pipeline configuration in response to changing edge network conditions.

5.2.1 Distributed Inference Runtime

The inference runtime uses **vLLM** [6] and **Ray** Clusters [16] to manage the distribution of the LLM through all the devices in the network. We leverage Pipeline Parallelism (PP) where the L layers of the model are divided into K contiguous shards. Each shard is assigned to a specific GPU rank. Communication between different ranks are managed using point-to-point communication primitives (Send/Recv) forming a sequential execution chain, in which a rank can only communicate with the previous and next rank.

5.2.2 The Agentic LLM-Driven Orchestrator

The agentic orchestrator behaves as the "brain" of the system. It maintains updated a comprehensive **connectivity map** that tracks the latencies between the nodes in the pipeline system alongside a **reasoning engine** powered by a Large Language Model and a **configuration manager** that updates the vLLM deployment settings based on the output received by the reasoning engine.

To enable a good reasoning the framework must have a real-time view of the environment, particularly it must manage a connectivity map containing the **inter-node latencies**, those are measured using ICMP Echo Requests (ping) to determine the Round-Trip Time (RTT) between sequential pairs of nodes, furthermore the decision engine must know all the available resources on the nodes including GPU models and memory specifications needed to determine maximum numbers of layers and KV cache capacity each node can host.

In our experimental testbed we leverage the Linux Traffic Controller (tc) tool to simulate varying network conditions, allowing us to validate the orchestrator's ability to react at rapidly changing network conditions.

5.2.3 Connectivity Map Example

The table 5.1 illustrates a typical edge network scenario where latencies are not all the same, for example link between **Node 1** and **Node 2** shows a big latency spike, while a static approach will continue to distribute layers equally (e.g "8 - 8 - 8 - 8"), our Agentic LLM-driven orchestrator might interprets the value as a bottleneck.

While reasoning on these values the agent can decide to move some layers from the 2 involved nodes or consolidate the pipeline into less nodes to skip the high-latency connection.

Table 5.1: Example Connectivity Map: Inter-node RTT (ms) as seen by the Orchestrator

Source \Dest	**Node 0**	**Node 1**	**Node 2**	**Node 3**
Node 0	0.0	2.2	5.1	8.4
Node 1	2.2	0.0	30.1	10.1
Node 2	5.1	30.1	0.0	10.5
Node 3	8.4	10.1	10.5	0.0

5.3 The Agentic Reasoning and Prompt used

The core innovation of this research is using a Large Language Model to solve the partitioning challenge. Unlike traditional solutions based on Integer Linear

Programming (ILP), which rely on rigid assumptions, the LLM-based agent can interpret a complex and large context, for example recognizing that a really slow link is the limiting factor for a real-time interaction.

5.3.1 Prompt Structure

The measured metrics in the connectivity map is transformed into a natural language prompt. The prompt includes:

- **Model Characteristics:** Model name, number of parameters, number of transformer layers and maximum context length;
- **System Constraints:** Number of nodes and resource available on each of them including the GPU model;
- **Guidelines:** Instructions to prioritize the evaluation metrics and avoid nodes where communication time exceeds the computational gain;
- **Output Format:** A requirement to provide the layer split in a given format (e.g. "A, B, C, D")

5.3.2 Prompt Example

Below we can see an example of the prompt used for our experiments with some sample model characteristics:

```
"You are a decision-making agent tasked with
configuring pipeline parallelism for inference of a
model using VLLM.
Current system configuration:
- Total nodes available: N
- Each node has x GPU: GPU Model
- Latency configuration between nodes (in ms):
   [a,b] [c],
  - First value: rank A -> rank B
  - Second value: rank B -> rank C
  - Third value: rank A -> rank C
Model: Model_Name
Architecture:
Transformer-based causal decoder-only architecture
X decoder blocks
A context length
Objective: Split the layers of the model across nodes to
```

```
optimize the following metrics while respecting constraints
Metrics to optimize:
1. **Output token throughput (tok/s)**
2. **Time to First Token (TTFT, ms)**
3. **Time per Output Token (TPOT, ms)**
4. **Inter-token Latency (ITL, ms)**
Constraints:
- Each node has finite GPU memory. Do not assign more
layers than a GPU can handle.
- Adding nodes to the pipeline is only beneficial if it
improves throughput without significantly increasing latency.
- Consider both intra-node computation times and inter-node
communication latency when deciding layer splits.
Task:
1. Analyze the optimal number of nodes to use. You may
choose to use fewer than X nodes if it improves overall performance.
2. Propose a layer-to-node assignment strategy for
pipeline parallelism.
3. Justify your reasoning considering memory constraints,
GPU processing times, and inter-node latency.
Format your response as:
- **Nodes used:** N
- **Layer assignment:** Node 0: layers [...], Node 1:
layers [...], ...
- **Rationale:** Explain why this configuration is optimal
considering latency, throughput, and memory limits."
```

5.3.3 Agent Autonomy in Managing Latency and Node Participation

A feature of this framework is that the latency threshold to consider a link not acceptable isn't hardcoded. Instead the LLM has full autonomy to decide the configuration while considering the trade-off between **computational gain** (using one more GPU in the pipeline) and **communication cost** (consider the latency added by one more network link), assigning also a different numbers of layers to each rank.

The LLM recognizes that in Pipeline Parallelism a slow link in the communication chain can affect negatively the maximum possible throughput. If a prompt indicates a link with a big latency, the agent evaluates if the time saved by having computation across more devices is smaller than the "penalty" introduced by the slow link. If the penalty is too high and the model can run without it, the agent assigns **zero**

layers to the slow node, intentionally excluding it and reducing the size of the distributed system to reduce the end-to-end latency.

5.3.4 Different Models Reasoning

Our experiments has been conducted using different "Orchestrator models" (e.g. Gemini 3, ChatGPT 5.1) revealing different optimization behaviors even when provided with the same latency maps. Those variability prove the agentic nature of our framework. Rather than solving a fixed optimization problem the system leverages the model's reasoning capabilities about distributed systems, as a result different models may reach at **different strategies**, with some of them producing more effective configurations than others.

5.4 Dynamic Pipeline Reconfiguration

Once the LLM-driven agent provides the optimal pipeline split and the configuration differs from the current one, the framework triggers a reconfiguration phase:

- **Pause** the incoming requests;
- **Redistributes** the model weights, re-initializes the inference engine and re-allocate the KV-Cache on each node;
- **Resumes** inference with the new pipeline configuration.

This capability allows the system to maintain an always optimize throughput even when the edge network fluctuates.

Chapter 6

Evaluation

This chapter evaluates the performance of the proposed agentic framework through a series of experiments on the Fabric Testbed [15]. We first we introduce the evaluation metrics used to evaluate the system performance. After, we describe the multi-node testbed configuration and analyze the benchmarking results obtained across various simulated network conditions through Linux Traffic Control (tc) and using Llama-2 and Falcon models.

6.1 Evaluation Metrics

Evaluating the performance of Large Language Model inference system requires some metrics (see Figure 6.1) able to capture both system efficiency and user-seen latency. New inference engines such as vLLM expose several metrics that describe the system behavior and how the distributed system is performing in many phases of the generation process, particularly during prefill phase and autoregressive inference.

In this thesis we evaluate the system using four well-known metrics: **Time To First Token (TTFT)**, **Time Per Output Token (TPOT)**, **Inter-Token Latency (ITL)** and **Output Throughput**. These metrics provide a large view of inference performance by capturing the inital latency, the generation speed and the maximum system capacity. The time metrics are usually measured in milliseconds (ms) and the throughput is measured in tokens per seconds (tok/s).

6.1.1 Time To First Token

Time To First Token (TTFT) measures the latency between the moment when a request is submitted and the generation of the first output token. This metric is largely determined by the cost of the **prefill** phase, in which the model processes

the entire input prompt before computing the initial attention.

Formally:

$$TTFT = t_{first\_token} - t_{request}$$

TTFT is really important for interactive applications such as chatbots or other real-time assistants, in which the user expects a fast response. Since this metric is influenced by the prefill phase the input length is the main affecting factor on the result.

6.1.2 Inter-Token Latency (ITL)

Inter-Token Latency (ITL) describes the interval time between two consecutive generated tokens during the autoregressive decoding phase.

$$ITL_i = t_{token_i} - t_{token_{i-1}}$$

This metric measures the generation speed of the model when the decoding phase has started, a lower ITL corresponds to faster token generation. This metrics is highly influenced by the concurrency of the inference system, especially in distributed environments.

6.1.3 Time Per Output Token

Time Per Output Token (TPOT) represents the average time required to generate the tokens after the first one. TPOT is calculated as the avarage of all inter-token latencies in the request:

$$TPOT = \frac{\sum ITL_i}{N_{tokens} - 1}$$

Where N_{tokens} represents the total number of generated tokens. A lower TPOT value corresponds to faster streaming responses.

6.1.4 Output Throughput

The **Output Throughput** measures the number of output tokens generated by the system per second across all the concurrent requests:

$$\text{Output Throughput} = \frac{\text{Total Generated Tokens}}{\text{Total Time}}$$

6.1.5 Metrics Consideration

Achieving a good balance between the previously described metrics is essential to provide the best end-user experience. For this reason these metrics shouldn't be evaluated independently, as optimizing a single one may negatively impact the others. Particularly in distributed environments the communication latency and the pipeline partitioning can affect each metrics differently and so selecting an appropriate deployment configuration requires evaluating the best trade-off between the obtained results. The goal of our thesis is to address this challenge by analyzing the current latency distribution between nodes and dynamically select a configuration that provides the **best overall balance** of the metrics, this is achieved by leveraging the reasoning capabilities of modern LLMs, which are used as decision-making agent to evaluate the system conditions and suggest the most efficient pipeline configuration, introducing a novel agentic LLM-driven framework.

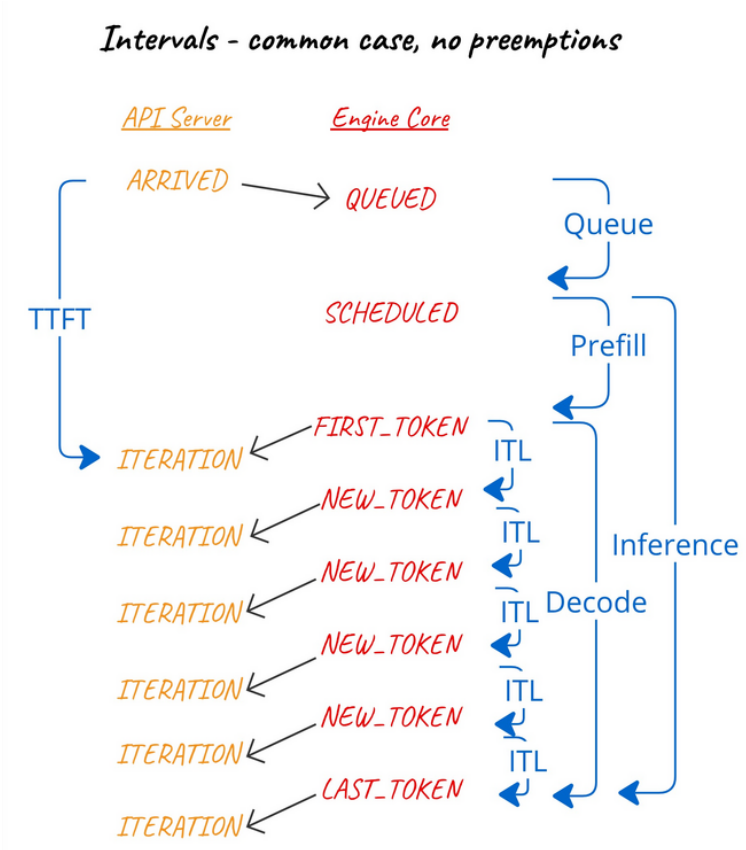

Figure 6.1: Metrics and intervals illustration. [17]

6.2 Experimental Results and Discussion

To evaluate the agent, we used Falcon3-10B-Base across a 3-node cluster. We simulate **different network conditions** using tc across the two links, denoted as $[L_1, L_2]$ representing respectively the added milliseconds latency between Node 0 to Node 1 and between Node 1 to Node 2. In this section we analyze the performance of the Falcon3-10B-Base model across various network latency configurations.

Compared to the our first approaches we decided in these experiments to use lower latency configurations (e.g. [2,4] ms). Even at these low scales the agents consistently find a way to optimize the configuration. To show the performances of different LLM reasoning engine some graphs are show, the experiments were conducted on different latencies configuration and run multiple times to obtain trusty confidence intervals. The LLM used for our decision making agents are Gemini 3 Reasoning and ChatGPT 5.1, those are compared with the default selection with 3-nodes of vLLM (equal split on every node) and a 2-nodes split to have some good benchmark comparisons.

6.2.1 Testbed Setup

The testbed configuration for these experiments consists in a 3-node cluster (see Figure 6.2, each node has an Nvidia Tesla T4 GPUs. Those Tesla GPUs provide 16 GB of VRAM, for a 10B parameter model like Falcon3 this results to a limited memory budget. In fact a 2-node setup with 20 layers assigned to each device pushes the limit of the available memory capacity. This constraint highlight also why the agent must assign weighs carefully and not just look to the latency measurements, since assigning too many layers to a node would lead to a configuration mismatch and result in a out-of-memory error.

6.2.2 Analysis on the number of nodes

Analyzing the Figure 6.3 we can find a primary observation regarding the impact of node number on the **Time to First Token (TTFT)**. In the optimal network conditions with zero latencies ([0,0]) the 3-node configurations (chosen also by Gemini 3) achieve a TTFT of approximately 2941 milliseconds. In contrast, the 2-node configuration (chosen also by ChatGPT 5.1) sees this value bounce up to 4571 ms, an increase of over 55%. This overhead is primarily caused by the increased computational load per node during the prefill stage, explained before. Using only 2 nodes with this model is technically feasible with our VRAM capacity but as we can see in our 2-nodes benchmarks there is no gain in the output throughput or ITL at [0, 0] ms latency (see Figures 6.6 and 6.4). So, in stable low-latency conditions, maximizing the available compute nodes is the better strategy.

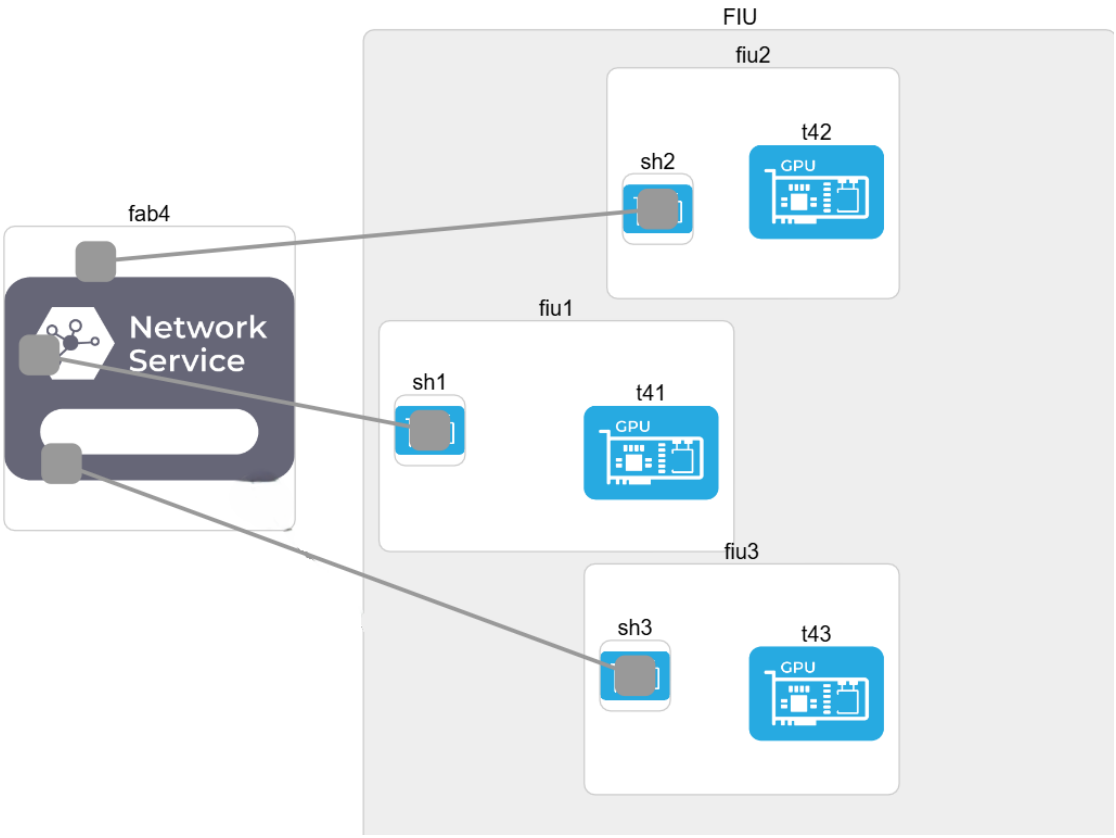

Figure 6.2: Fabric Testbed Configuration Used for Falcon3-10B-Base experiments.

6.2.3 Sacrificing TTFT for ITL

One of the most interesting result of this evaluation appears as the network latency increases, for example looking at the **[2, 10] ms** configuration. Under those conditions we can see that the adopted split strategies start to diverge significantly:

- The slow 3-nodes: the default split (13/14/13) shows that while it maintains a lower TTFT (~3617 ms), **the Inter-Token Latency (ITL) spikes to ~161 ms** and the throughput drops to 47.39 tok/s. This is because every single generation step must traverse the 10 ms link.
- The bypass of 2-nodes: ChatGPT 5.1 opts for a symmetric 20/20 split, bypassing the third node that would introduce latencies. While TTFT is signifcantly higher (~4851 ms), the **ITL is reduced to ~118 ms** while the throughput is up to almost 60 tok/s.

From a User Experience (UX) perspective this is a critical revealing. While a user may tolerate an extra second of waiting for the response starting displaying on

screen (TTFT), a high ITL (slow generations of words) is much more disturbing. Our results suggest that at higher latencies it's more beneficial to **strategically avoid** slower nodes to minimize the communication penalty for each generation step.

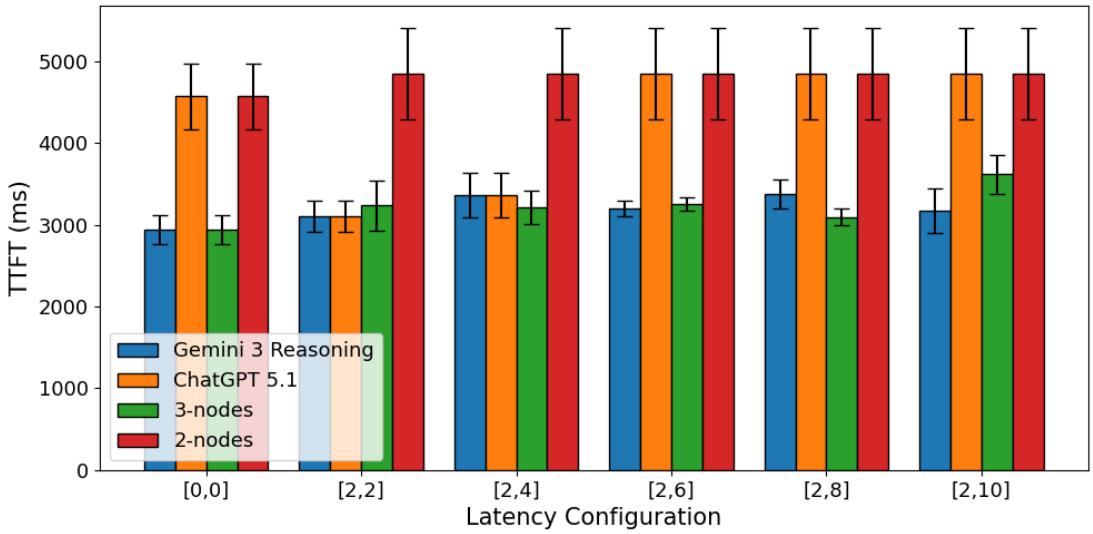

Figure 6.3: TTFT plot for Falcon3-10B-Base. It represents the waiting time before the first generated token. On 2-nodes it's usually longer due to the reduced total computation power available.

6.2.4 Different Agents Reasoning Capabilities

The comparison between ChatGPT 5.1 and Gemini 3 Reasoning shows the varying way of thinking of the modern reasoning models that can be used in our agentic orchestration for solving distributed inference problems:

- **Gemini 3 Reasoning:** Tends to act more conservative, it consistently maintain a 3-node setup, even at the maximum latency configuration, by slightly adjusting layers between nodes. This approach optimizes TTFT but it fails to recognize the huge throughput gains that would happen by dropping one slow node.
- **ChatGPT 5.1:** Shows a more aggressive optimization logic. From a "moderate" configuration ([2,6]) it starts to switch to a 2-node configuration. This approach achieves the best throughput and ITL in those stressed conditions.
- Even if ChatGPT 5.1 reached the "best" decision for high latency, it's not

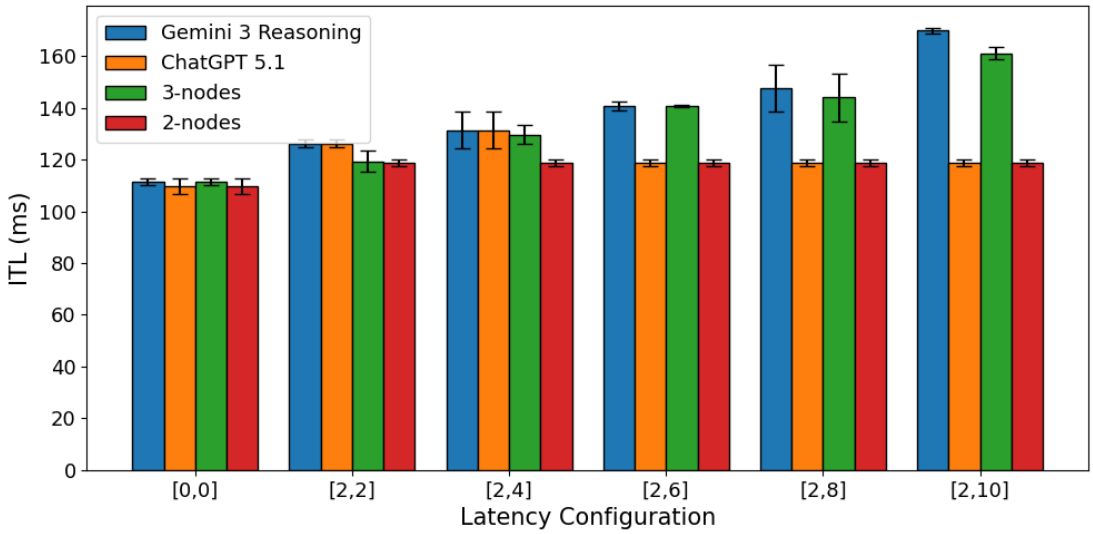

Figure 6.4: ITL plot for Falcon3-10B-Base. It represents the time between each token generation, while the initial wait (TTFT) might be higher on fewer nodes the lower ITL on the same latency configuration compensate by providing a much faster and more stable token stream

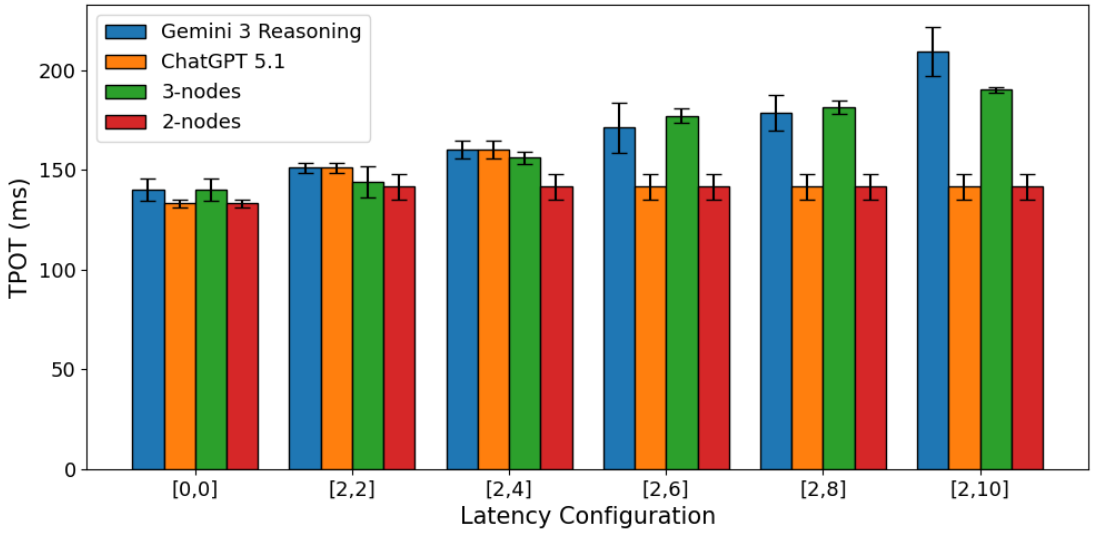

Figure 6.5: TPOT plot for Falcon3-10B-Base. It represents the average processing time required for each output token, it's highly related to ITL. The agentic orchestrator stabilizes this metric by bypassing slow network links

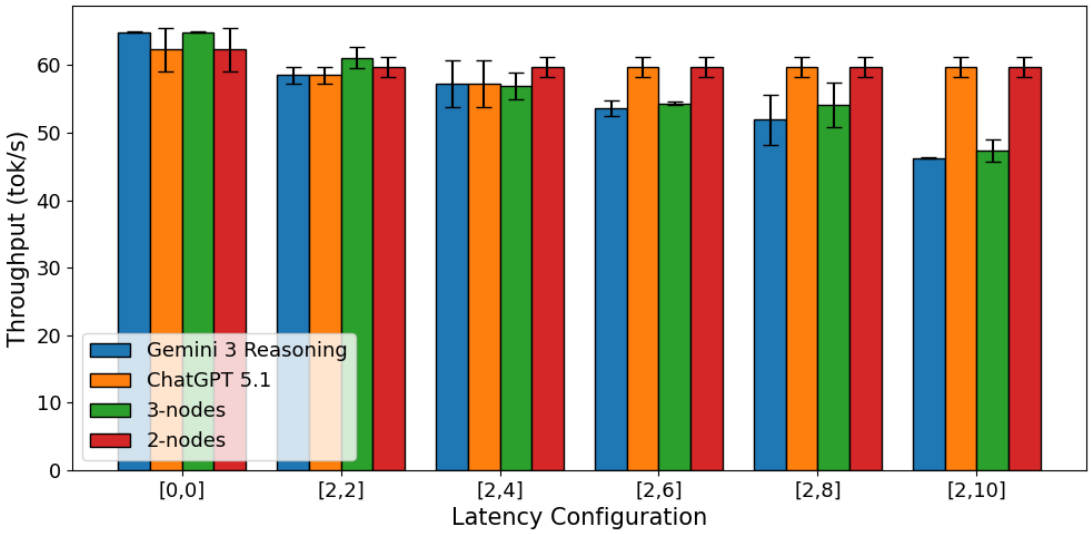

Figure 6.6: Output Throughput plot for Falcon3-10B-Base. It represents the overall output rate of tokens generation. The result show that in stressed network scenarios, a 2-nodes configuration, and so the agentic orchestrator selecting it, achieves an higher throughput by eliminating the communication bottleneck, justifying the trade-off of having a slower initialization time

always perfect. In lower latency configurations it might not find the best solution, as we can see in the graphs.

From a reasoning perspective the two models approach the optimization problem using different assumptions, **Gemini** tends to take a **performance evaluation** strategy, it estimates the computational cost of each transformer layers on the GPU and calculates how distributing those layers across the multiple pipeline devices reduces the bottleneck, based on this it prefers to keep all the available nodes and compensate the communication latency by rebalancing the numbers of layers assigned to each device. On the other hand **ChatGPT** tends to reason more about the **network topology and the cost of communication** between nodes, it doesn't estimate the computation time but instead it prioritizes minimizing the number of network hops when the links latency become large, this is why after the [2, 6] configuration it starts to exclude the slow link to adopt a two-node pipeline configuration.

As we thought, this shows that while LLMs are very capable to reason about distributed systems, their performance is sensitive to the context provided. We believe that with further **Prompt Engineering**, these models can reach the always best configuration more often, for example by providing historical configuration data with their relative metrics.

Latency (ms)	Model	Assignment	ITL (ms)	TTFT (ms)	TPOT (ms)	(Token/s)
[0, 0]	Gemini 3 Reasoning	13 / 14 / 13	111.49	2941.22	140.14	64.90
[0, 0]	ChatGPT 5.1	20 / 20	109.72	4571.30	133.20	62.31
[0, 0]	3-nodes	13 / 14 / 13	111.49	2941.22	140.14	64.90
[0, 0]	2-nodes	20 / 20	109.72	4571.30	133.20	62.31
[2, 2]	Gemini 3 Reasoning	13 / 13 / 14	126.33	3100.82	151.12	58.52
[2, 2]	ChatGPT 5.1	13 / 13 / 14	126.33	3100.82	151.12	58.52
[2, 2]	3-nodes	13 / 14 / 13	119.38	3235.20	144.09	61.10
[2, 2]	2-nodes	20 / 20	118.83	4851.82	141.59	59.73
[2, 4]	Gemini 3 Reasoning	13 / 13 / 14	131.39	3364.27	160.41	57.29
[2, 4]	ChatGPT 5.1	13 / 13 / 14	131.39	3364.27	160.41	57.29
[2, 4]	3-nodes	13 / 14 / 13	129.72	3215.16	156.14	56.93
[2, 4]	2-nodes	20 / 20	118.83	4851.82	141.59	59.73
[2, 6]	Gemini 3 Reasoning	15 / 12 / 13	140.78	3200.78	171.24	53.66
[2, 6]	ChatGPT 5.1	20 / 20	118.83	4851.82	141.59	59.73
[2, 6]	3-nodes	13 / 14 / 13	140.91	3249.14	177.18	54.32
[2, 6]	2-nodes	20 / 20	118.83	4851.82	141.59	59.73
[2, 8]	Gemini 3 Reasoning	16 / 11 / 13	147.76	3375.91	178.89	51.89
[2, 8]	ChatGPT 5.1	20 / 20	118.83	4851.82	141.59	59.73
[2, 8]	3-nodes	13 / 14 / 13	144.04	3093.33	181.53	54.13
[2, 8]	2-nodes	20 / 20	118.83	4851.82	141.59	59.73
[2, 10]	Gemini 3 Reasoning	17 / 9 / 14	169.90	3167.40	209.52	46.23
[2, 10]	ChatGPT 5.1	20 / 20	118.83	4851.82	141.59	59.73
[2, 10]	3-nodes	13 / 14 / 13	161.20	3617.34	190.46	47.39
[2, 10]	2-nodes	20 / 20	118.83	4851.82	141.59	59.73

Table 6.1: Performance Results Across Latency Configurations with Falcon3-10B-Base

Chapter 7

Conclusions

This final chapter brings what we saw together, we successfully identify the challenges introduced by distributed inference regarding the communication and we proved that an LLM-driven agent can act as a revolutionary "brain" to get a good pipeline configuration to balance and fix the fluctuating network conditions.

7.1 Contributions Summary

This thesis addressed the critical challenge of deploying Large Language Models (LLMs) across heterogeneous and fluctuating edge environments. Instead of relying on static partitioning strategies, we introduced an **Agentic LLM-driven Orchestration Framework** that treats distributed inference as a dynamic decision-making process rather than a fixed configuration.

The main contributions of this work are:

- **Bottleneck analysis:** through our profiling work using the testbed, we showed that Pipeline Parallelism can spend up to 60% of the execution time in "bubbles" where communication isn't happening and this can percentage can increase in case of high network latencies;
- **Decoupled Architecture:** we proposed a modular design that separates the LLM-driven orchestrator (the reasoning agent, responsible for planning and coordination) from the high-performance distributed inference runtime (responsible for executing model computations). This separation enables high-efficiency system reconfiguration as the network conditions change, ensuring the decision-making logic remains independent from the execution engine;
- **Strategic nodes bypass:** we demonstrated that the act of **strategically avoiding** slower nodes can significantly improve user-perceived system behavior by reducing ITL (Inter-Token Latency).

7.2 Key Findings

Our experiment results with Falcon3-10B-Base model showed that **more compute nodes doesn't always lead to having better performances**. In stable and fast networks, a full-cluster deployment with 3 nodes computation provides the best balance of memory and system speed. However, as network conditions degrade the time spent on communication quickly outweighs the time gained with fast computation. Our agentic approach showed also that sacrificing Time to First Token (TTFT) to achieve a lower Inter-Token Latency (ITL) is a better strategy for having a smooth text stream. The comparison between different LLM orchestrator, Gemini 3 Reasoning and ChatGPT 5.1, showed that while these models can "think" having a full view on what the system is at every moment they reveal distinct optimization styles. This highlights that the intelligence of the chosen orchestrator model is as critical as the system computation capacity.

7.3 Limitations and Future Work

While this thesis proposed framework represents a significant step toward an autonomous and self-adjusting AI edge infrastructure, several directions for future research remain:

- **Advanced Prompt Engineering:** During our tests we saw that the agents sometimes over-correct. Providing the model with some standard examples or implementing the "Chain-of-Thought" prompting [18], a novel approach to guide a model through a step-by-step reasoning process by generating intermediate results, could enhance the decision-making process.

- **Multi-Agent Collaboration:** In a massive edge deployment, where we can find 10 or more nodes, a single LLM may struggle to find a good deployment solution, since the complexity increase rapidly. A multi-agent system, where small agents report the results to a global orchestrator, could achieve a better scalability.

- **Fine-Tuned Orchestrator LLM**: To resolve the different "personalities" and inconsistencies between different LLMs, a specialized model could be trained. A future research can be to fine-tune an "Orchestrator LLM" for our Agentic LLM-driven Framework, trained on system performance logs, to have a more accurate decision-maker.

7.4 Concluding Remarks

Transitioning from a static assignment system to an agentic LLM-driven system marks a shift in how we view distributed inference. We give the system the "autonomy" to reason about its own bottlenecks, moving closer to resilient AI system that can close the gap between cloud infrastructure and the unpredictable constraints of real-world edge environments.

Bibliography

[1] Fanlong Zeng, Wensheng Gan, Yongheng Wang, and Philip S Yu. «Distributed training of large language models». In: *2023 IEEE 29th international conference on parallel and distributed systems (ICPADS)*. IEEE. 2023, pp. 840–847 (cit. on pp. 5, 19).

[2] Minoo Hosseinzadeh and Hana Khamfroush. «DILEMMA: Joint LLM quantization and distributed LLM inference over edge computing systems». In: *arXiv preprint arXiv:2503.01704* (2025) (cit. on pp. 5, 6, 8).

[3] Kai Zhang, Hengtao He, Shenghui Song, Jun Zhang, and Khaled B Letaief. «Distributed on-device LLM inference with over-the-air computation». In: *ICC 2025-IEEE International Conference on Communications*. IEEE. 2025, pp. 6291–6296 (cit. on pp. 6, 8).

[4] Mingjin Zhang, Xiaoming Shen, Jiannong Cao, Zeyang Cui, and Shan Jiang. «Edgeshard: Efficient llm inference via collaborative edge computing». In: *IEEE Internet of Things Journal* 12.10 (2024), pp. 13119–13131 (cit. on pp. 7, 8).

[5] Benedetta Picano, Dinh Thai Hoang, and Diep Nguyen. «A matching game for LLM layer deployment in heterogeneous edge networks». In: *IEEE Open Journal of the Communications Society* (2025) (cit. on pp. 7, 8).

[6] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. «Efficient memory management for large language model serving with pagedattention». In: *Proceedings of the 29th symposium on operating systems principles*. 2023, pp. 611–626 (cit. on pp. 9, 25).

[7] Yuhan Liu et al. «Cachegen: Kv cache compression and streaming for fast large language model serving». In: *Proceedings of the ACM SIGCOMM 2024 Conference*. 2024, pp. 38–56 (cit. on p. 9).

[8] Zeyu Zhang, Haiying Shen, Shay Vargaftik, Ran Ben Basat, Michael Mitzenmacher, and Minlan Yu. «Hack: Homomorphic acceleration via compression of the key-value cache for disaggregated llm inference». In: *Proceedings of the ACM SIGCOMM 2025 Conference*. 2025, pp. 1245–1247 (cit. on pp. 9, 10).

[9] Yuhan Liu et al. «DroidSpeak: KV Cache Sharing for Cross-LLM Communication and Multi-LLM Serving». In: *arXiv preprint arXiv:2411.02820* (2024) (cit. on p. 10).

[10] Balkiss.hamad. *Apprentissage Supervisé Vs Non Supervisé (Supervised vs Unsupervised Learning)*. Derived version translated to English by Alenoach. Wikimedia Commons. n.d. URL: `https://commons.wikimedia.org/w/index.php?curid=155627022` (cit. on p. 12).

[11] Fangfang Lee. *What is a neural network?* `https://www.ibm.com/think/topics/neural-networks`. Accessed: January 11, 2026 (cit. on p. 12).

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. «Attention is all you need». In: *Advances in neural information processing systems* 30 (2017) (cit. on pp. 13, 14).

[13] Josh Levy-Kramer. *Demystifying Tensor Parallelism*. `https://robotchinwag.com/posts/demystifying-tensor-parallelism`. Accessed: January 11, 2026 (cit. on p. 16).

[14] Susan Zhang et al. «Opt: Open pre-trained transformer language models». In: *arXiv preprint arXiv:2205.01068* (2022) (cit. on p. 18).

[15] Ilya Baldin, Anita Nikolich, James Griffioen, Indermohan Inder S Monga, Kuang-Ching Wang, Tom Lehman, and Paul Ruth. «FABRIC: A national-scale programmable experimental network infrastructure». In: *IEEE Internet Computing* 23.6 (), pp. 38–47 (cit. on pp. 18, 30).

[16] Philipp Moritz et al. «Ray: A distributed framework for emerging {AI} applications». In: *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 2018, pp. 561–577 (cit. on p. 25).

[17] vLLM. *Metrics*. `https://docs.vllm.ai/en/stable/design/metrics/`. Accessed: January 11, 2026 (cit. on p. 32).

[18] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. «Chain-of-thought prompting elicits reasoning in large language models». In: *Advances in neural information processing systems* 35 (2022), pp. 24824–24837 (cit. on p. 40).