UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

UNIVERSITY OF MODENA AND REGGIO EMILIA

"Enzo Ferrari" Department of Engineering

Master's Degree in Artificial Intelligence Engineering (LM-32)

Development of PEFT-Based Extraction Methodologies for Small-Scale LLMs

Supervisor:
Prof. Lorenzo Baraldi

Candidate:
Giacomo Gherardini
Student ID 207709

ACADEMIC YEAR 2025/2026

Contents

List of Figures

List of Tables

List of Codes

List of Abbreviations

AI	Artificial Intelligence
BERT	Bidirectional Encoder Representations from Transformers
BF16	Brain Floating Point 16
BLEU	Bilingual Evaluation Understudy
CIDEr	Consensus-based Image Description Evaluation
CLM	Causal Language Modeling
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSV	Comma-Separated Values
EOS	End Of Sequence
ERP	Enterprise Resource Planning
FFN	Position-wise Feed-Forward Network
FFT	Full Fine Tuning
FP16	Float16
GPU	Graphic Processing Unit
HPC	High Performance Computing
JSON	JavaScript Object Notation
JSONL	JavaScript Object Notation Line
LLM	Large Language Model
LoRA	Low-Rank Adaptation
LSTM	Long Short-Term Memory
METEOR	Metric for Evaluation of Translation with Explicit Ordering

MLM	Masked Language Modeling
MLP	Multi-Layer Perceptron
MMLU	Massive Multitask Language Understanding
NF4	NormalFloat4
NLP	Natural Language Processing
OCR	Optical Character Recognition
OOM	Out Of Memory
PDF	Portable Document Format
PE	Positional Encoding
PEFT	Parameter Efficient Fine Tuning
PPO	Proximal Policy Optimization
QLoRA	Quantized Low-Rank Adaptation
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RLHF	Reinforcement Learning with Human Feedback
RNN	Recurrent Neural Network
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
ROUGE-L	ROUGE Longest Common Subsequence
Seq2Seq	Sequence-to-Sequence
SFT	Supervised Fine-tuning
SQL	Structured Query Language
SSH	Secure Shell
TF-IDF	Term Frequency-Inverse Document Frequency
UTF-8	8-bit Unicode Transformation Format
ViT	Vision Transformer
VRAM	Video Random Access Memory

1. Introduction

The rapid evolution of LLMs in recent years has revolutionized the field of Natural Language Processing (NLP), enabling unprecedented capabilities in text generation, reasoning, and data processing. LLMs are trained on massive amounts of general purpose data, allowing them to demonstrate remarkable learning capabilities across a broad spectrum of generic tasks, even achieving "zero-shot learning". However, when it comes to using an LLM for specific, customized tasks, this can represent a paradigm shift that the model itself struggles to adapt to. When dealing with proprietary data, strict technical jargon, or standardized documents, the basic LLMs frequently fail to provide the required level of accuracy. They often lack the in-depth, domain-specific knowledge needed to correctly and comprehensively interpret the specific information we provide them, leading to inaccurate or generic results that are unsuitable for the applications the user needs.

Consequently, to achieve high reliability in specific tasks, an LLM must undergo a process of specialization, adapting its vast pre-trained knowledge to perform highly specific downstream tasks. Yet, fully specializing a model requires immense computational power and memory, creating a prohibitive financial and infrastructural barrier for many small to medium-sized enterprises. Overcoming this bottleneck requires the adoption of training methodologies capable of democratizing access to Artificial Intelligence (AI).

This thesis addresses these challenges by studying the application of PEFT techniques to extract structured information from technical documents. Specifically, the research focuses on the industrial mechanical sector, with the goal of automating the extraction of technical specifications of mechanical components directly from Portable Document Format (PDF) documents illustrating the objects and their characteristics.

Before explaining the thesis project, chapter two provides the theoretical foundations of LLM. It explores the evolution of the Transformer architecture, including attention mechanisms and vision variations, explains the fundamental differences between causal and masked training methodologies, and thoroughly details the PEFT techniques, including LoRA and QLoRA, that represent the core mathematical engine of this research.

Subsequently, the third chapter explains how the thesis project was structured to achieve the fine-

tuning objectives for the LLMs. The project was organized into four main operational phases. The first phase involved the creation of a custom ground truth dataset from scratch. Approximately twenty thousand PDF documents representing mechanical components were collected and curated, filtering them from a massive raw repository of approximately one hundred thousand total documents. In the second phase, a data extraction pipeline was developed using Python scripts, native libraries, and Tesseract OCR to process both the visual layout and the textual data. The extracted information, such as component type, physical description, and categorization, was then organized into a standardized JSONL format, making it structurally sound and suitable for supervised training.

After data preparation, the research moved on to the model fine-tuning phase. To address the computational constraints typical of many business environments, the study deliberately focused on small-scale architectures. A selected group of recent open-weight models was used, including representatives of the Llama, Mistral, Qwen, and Phi families. A structural comparison was then established by fine-tuning models with seven and fourteen billion parameters to evaluate performance across different scales. Training was executed on High Performance Computing (HPC) environments: preliminary implementation tests were run on a cloud-based platform called Runpod to verify the cost-effectiveness for small private companies, while the core thesis experiments were conducted entirely on the Unimore HPC cluster using industrial-grade Graphic Processing Units (GPUs), specifically the NVIDIA A40 GPU. A key aspect of this phase was the execution of two parallel training iterations for each chosen model: one using standard LoRA and the other using QLoRA.

The final phase focused on inference testing and evaluation. By monitoring the validation loss during training and applying standard metrics such as BLEU, METEOR, CIDEr, and ROUGE, the models' ability to accurately extract and generate the required technical data was assessed. Furthermore, the final results aim to provide insights into three critical research questions: the evaluation of the scaling law through the performance comparison of different parameter sizes, the measurement of the effective performance cost introduced by LoRA and the 4-bit quantization in QLoRA, and the definition of a direct comparison between the architectures with the best performance in terms of precision.

2. Theoretical Foundations of LLMs

This chapter introduces the fundamental concepts and architectures that form the basis of modern LLMs. It is essential to understand how these models process human language, how they are trained, and how they have evolved to achieve their current reasoning capabilities. The discussion begins with the core architecture, the **Transformer**, moves on to training methodologies, and concludes with PEFT strategies.

2.1 Transformers

The architecture on which modern LLMs are generally based is Transformer, an architecture created and introduced by Google Brain researchers in 2017 with the seminal paper "*Attention Is All You Need*" [19]. Originally designed to solve complex translation tasks, it subsequently radically changed the paradigm of sequential processing.

Before its introduction, the NLP landscape was dominated by models that read text sequentially, step by step. As we'll see in the next subsection, this traditional approach created significant computational bottlenecks and memory limitations. Transformer completely changed this paradigm, proposing a new architecture based on a mathematical operation called **attention**, completely eliminating sequential processing and its related bottlenecks [19].

Transformer's key feature is its ability to process entire sequences simultaneously. Given a source sequence $X = [x_1, x_2, ..., x_n]$ and a target sequence $Y = [y_1, y_2, ..., y_n]$, the attention operator allows the model to observe all elements simultaneously. This gives the network an infinite receptive field, meaning it can easily capture long-range dependencies and understand the deep context of a word regardless of its physical distance from other relevant words in the input.

By avoiding sequential constraints, Transformer is a parallelizable architecture. This architectural advantage dramatically reduces training time and allows it to scale to large datasets. Its utility has rapidly expanded beyond simple translation, becoming the undisputed state-of-the-art for complex language understanding, reasoning, and structured data extraction. Furthermore, advanced variants such as Graph Transformers have subsequently demonstrated strong capabilities in

modeling complex topological and structural dependencies that go beyond standard sequential text [22].

Structurally, the original Transformer is based on a bipartite framework: an **encoder** and a **decoder**. The **encoder** receives a sequence of raw input representations (the source sequence X) and maps it into a sequence of abstract, continuous mathematical representations, often denoted $Z = [z_1, ..., z_n]$. Once this contextualized representation is constructed, the **decoder** uses Z to generate the final output sequence Y, producing one element at a time [19].

While the Transformer adheres to this general macro-structure, it implements it using overlapping layers of **self-attention** and fully connected neural networks. In the following sections, we will retrace the evolutionary path that led to this architecture, starting from the limitations of the pre-Transformer era, and then analyze in detail the internal mathematical components that allow these models to achieve high performance.

2.1.1 The Pre-Transformer Era

Before the introduction of the Transformer architecture, the field of sequence modeling and NLP relied primarily on two types of neural networks: **Recurrent Neural Networks (RNNs)** and **Convolutional Neural Networks (CNNs)**. Although revolutionary for their time, both architectures had fundamental limitations that hindered their ability to process long, complex sequences efficiently.

RNNs, including variants such as Long Short-Term Memory (LSTM) networks, were explicitly designed to handle sequential data. Their operation is based on processing a sequence of input tokens, either token-by-token, sequentially from left to right or right to left. At each time step t, the network computes a hidden state s_t based on the current input x_t and the hidden state from the previous time step s_{t-1} [15]. This design theoretically allows the network to maintain a memory of past information. However, the sequential nature of this network presents both a physical and computational bottleneck: the current step's computation depends on the completion of the previous one [19]. As a result, this strict sequentiality completely precludes parallelization in training examples.

This limitation becomes a problem when dealing with long documents or when trying to fully exploit the parallel computing power of modern GPUs. Furthermore, RNNs face a problem known as the "vanishing gradient." During the training phase, the network learns by propagating the error

signal backward in time. For excessively long sequences, this error is multiplied by itself many times at each step. Since these mathematical values are often small fractions, repeated multiplication causes the signal to reduce exponentially until it "disappears." As a result, the network is unable to adjust its weights for the first few steps, and information presented early in the sequence is inevitably lost [15].

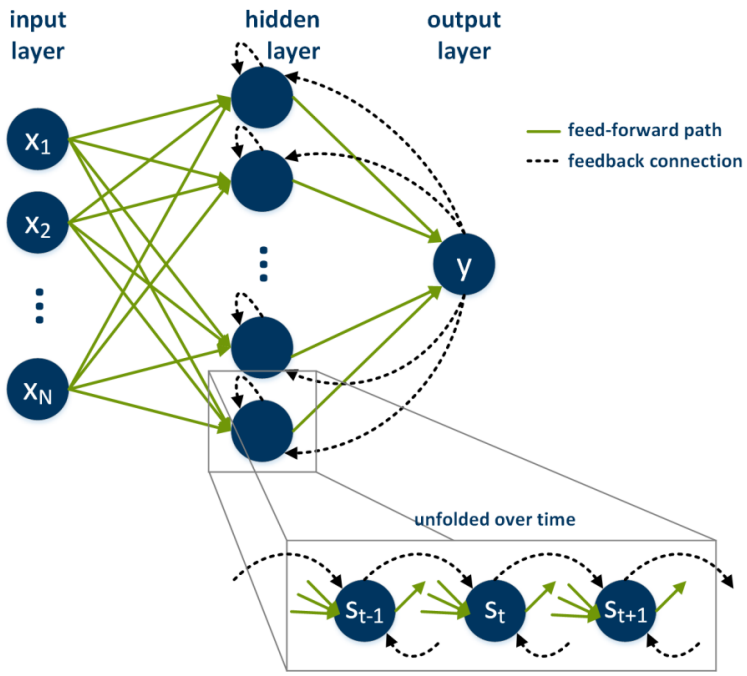

Figure 2.1: Principle of RNNs, layers are connected to their predecessors via feedback connections [8].

On the other hand, **CNNs** were initially popularized by their immense success in computer vision tasks, such as image classification [11]. In a CNN, mathematical operators, called convolutional filters, are applied to local regions of the input data [11]. Unlike RNNs, the CNNs can compute these local representations in parallel for all locations simultaneously, which represents a significant advantage in terms of computational efficiency [19].

For this reason, researchers have attempted to adapt CNNs to text and sequence processing. However, they have introduced a different, but equally serious, limitation related to their "receptive field." A single convolutional layer considers only a small window of adjacent tokens. To correlate signals from two distant locations in a sequence, a CNN requires a complex structure composed of multiple convolutional layers. The number of operations required to connect these tokens increases

depending on the specific architecture: it grows linearly ($O(n/k)$) for standard sequential CNNs or logarithmically ($O(\log_k(n))$) if the network uses "dilated" convolutions, which skip intermediate steps to more rapidly expand the field of view [19].

This architectural constraint makes it extremely difficult for the network to learn and capture long-range dependencies within a text.

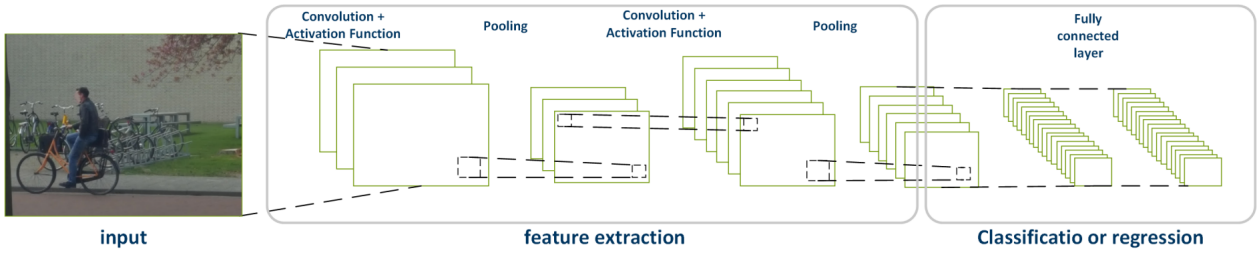

Figure 2.2: Common processing flow of a CNN [8].

Ultimately, the pre-Transformer era was characterized by a trade-off: one had to choose between the sequential memory of RNNs, sacrificing parallelization, or the parallel efficiency of CNNs, sacrificing the ability to easily connect distant concepts. Transformer was designed to simultaneously overcome both of these limitations, offering complete hardware parallelization while reducing the computation of the relationship between any two locations to a constant number of operations, equal to $O(1)$ [19].

2.1.2 From Sequence-to-Sequence to the Attention Mechanism

Before the advent of the Transformer, **Sequence-to-Sequence (Seq2Seq)** models were the standard approach for NLP tasks, such as machine translation. These models relied heavily on RNNs and LSTMs [17]. The core idea of a traditional Seq2Seq model was to use an encoding network to read the input sequence token by token and compress all its semantic meaning into a single fixed-length vector, often called a *context vector* [17]. A decoding network then took this single, continuous representation and expanded it to generate the output sequence, producing one word at a time.

Although logically sound, this approach has a major representational limitation: compressing a very long sequence into a single, fixed-size vector inevitably leads to information loss. This happens because the neural network is forced to represent every single nuance of the input in a static space. As a result, the model would forget the initial parts of the sequence once it reached the end, and its

performance would rapidly degrade as the length of the input sentences increased [19].

The **attention mechanism** was designed precisely to overcome this bottleneck. Instead of forcing the encoder to compress everything into a single static vector, the attention mechanism allows the decoder to "look back" at the entire input sequence, at each stage of the generation process. This allows for a link between the encoder and the decoder. Mathematically, the model assigns a different weight or *importance score* to each word in the source sequence, depending on the word it is currently trying to generate [19].

This infinite receptive field means that each token can pay attention to any other token in the sequence, regardless of their distance. By dynamically computing this weighted context at each step, the attention mechanism completely solved the long-range dependency problems that plagued standard RNNs and CNNs, laying the perfect foundation for the purely attention-based architecture that would soon follow [19].

2.1.3 Self-Attention and Multi-Head Attention

The attention mechanism in the Transformer architecture was developed to allow the network to dynamically determine the priority and relevance to assign to individual input tokens. This mathematical operation is applied to sequences of three distinct entities, drawing an analogy from database retrieval systems: **queries** Q, **keys** K, and **values** V [19].

The query represents what the current token is looking for, the key represents what another token in the sequence has to offer, and the value is the actual content of that token. To maximize computational efficiency, the vectors representing the individual tokens are grouped into large matrices. This allows the model to simultaneously calculate attention scores for all tokens using matrix multiplication, avoiding the sequential processing of previous architectures.

The shape of these matrices is determined by the length of the sequences and the internal dimensionality of the model. Specifically, the n values indicate the length of the sequences, i.e., the number of tokens. The d values represent the size of the vectors used to mathematically describe each token: neural networks do not process raw text; instead, they convert each token into a continuous array of numbers called $embedding$ [19]. The d dimensionality simply determines the length of this array.

Therefore, the resulting matrices are defined as follows:

$$Q \in \mathbb{R}^{n_q \times d_k}, \quad K \in \mathbb{R}^{n_k \times d_k}, \quad V \in \mathbb{R}^{n_k \times d_v} \tag{2.1}$$

Given these three matrices, the general attention function evaluates how well the queries match the keys. This similarity determines how much of the values should be passed forward to the next layer. The operation is mathematically defined as:

$$f(Q, K, V) = \text{softmax}(\alpha(Q, K^T))V \tag{2.2}$$

where α is a function specifically designed to compute the similarity between the matrices Q and K. In the standard Transformer architecture, this similarity is computed using a simple dot product. However, as the dimensions increase, the values of the dot product can become extremely large. This magnitude pushes the softmax function into regions where the gradients are exceptionally small, hindering the learning process during backpropagation [19]. To avoid this problem, the dot product is scaled by dividing it by the square root of the dimensionality of the key vectors, $\sqrt{d_k}$. This adjustment gives rise to the scaled dot-product attention, defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.3}$$

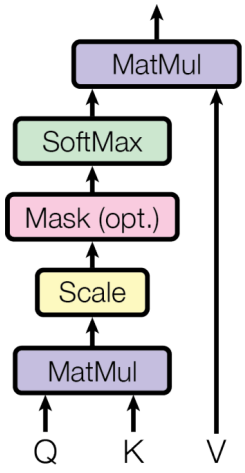

Figure 2.3: Illustration of the Scaled Dot-Product attention mechanism: computing the dot product between queries and keys, applying a scaling factor to stabilize gradients, and using a softmax function to determine the weights applied to the final values [19].

We define the output of this attention operator as out. Given the dimensions of the tensors defined previously, the resulting matrix naturally takes the form:

$$out \in \mathbb{R}^{n_q \times d_v} \tag{2.4}$$

For structural simplicity and to enable the creation of complex architectures, the dimensions for queries, keys, and values are often chosen identically ($d_k = d_v = d_{model}$). This way, the output of the attention mechanism maintains exactly the same dimensionality as its input. This design choice allows for seamless stacking of multiple transformer blocks.

This attention operator is primarily used in two ways within the architecture. In a **self-attention block**, queries, keys, and values are extracted from the same sequence via distinct learnable linear projections (W^Q, W^K, W^V). Mathematically, if we define X as the matrix representing the input sequence currently being processed, the model simply multiplies this input data by the projection weight matrices:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V \tag{2.5}$$

In contrast, in a **cross-attention block**, queries are extracted from one sequence, typically the generation output of the decoding layer mathematically represented by the matrix X, while keys and values are extracted from an entirely different sequence, typically the final output of the encoder represented by the matrix Y [19]. This formulation allows the generator sequence to selectively extract context from the original source sequence:

$$Q = XW^Q, \quad K = YW^K, \quad V = YW^V \tag{2.6}$$

However, relying on a single attention mechanism limits the model to focusing on only one specific type of relationship at a time. To overcome this limitation, the architecture extends the concept to a **Multi-Head Attention** mechanism [19]. Instead of running a single attention function on the full-dimensional tensors, the model generates h different heads of the queries, keys, and values. This is achieved by applying h independent linear projections to the original matrices Q, K, and V, projecting them into lower-dimensional subspaces.

The scaled dot-product attention is then applied to each of these h variants independently and in

parallel. Finally, the outputs from all the heads are concatenated together and passed through a final fully connected layer to restore the correct tensor shape. The power of this mechanism allows the model to simultaneously pay attention to different representation subspaces at different locations. Parallelizing this computation increases the representation capabilities of a single transformer layer while maintaining an efficient training process.

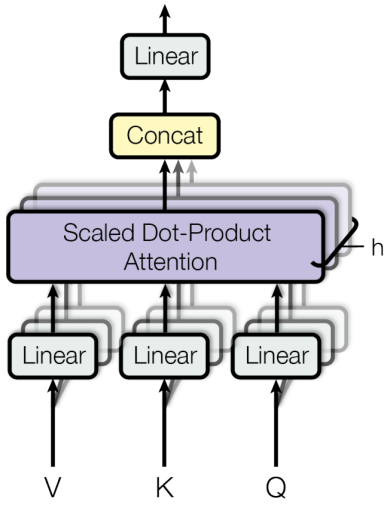

Figure 2.4: Illustration of the multi-head attention mechanism, showcasing the parallel computation of multiple attention heads and their subsequent concatenation [19].

2.1.4 Encoder-Decoder vs Decoder-Only

The original Transformer was introduced as an *Encoder-Decoder* architecture [19]. However, an input sequence of raw text cannot be directly fed into the mathematical operations of the network. It must first pass through an *Embedding Layer*: this layer translates discrete tokens (words or sub-words) into continuous, dense vectors of a fixed dimension (d_{model}), ensuring the input shape is compatible with the internal dimensionality of the model through a learnable linear projection [19].

During training, especially for generation tasks, a **masking mechanism** must be implemented. Transformers are designed to perform autoregressive generation: given a sequence of elements, the goal is to predict the next one, add it to the input sequence, and repeat. During training, the model is fed the entire sequence at once to optimize parallelism efficiency. However, it is crucial to ensure that when the model processes a specific token to predict the next one, it cannot simply cheat by looking ahead to future tokens already present in the training sequence [19].

This constraint is enforced in the decoder through a masking matrix applied to the attention scores before the softmax operation. The attention matrix is multiplied by a mask $\mathcal{M}$ to force the attention values associated with future tokens to negative infinity ($-\infty$), which the softmax function subsequently translates to zero [19]. Mathematically, the mask operates as a lower triangular matrix:

$$\mathcal{M} \in \mathbb{R}^{n \times n} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ 1 & 1 & 0 & \cdots & 0 & 0 \\ 1 & 1 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 1 & 1 & \cdots & 1 & 0 \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix} \tag{2.7}$$

To stabilize the training process, the architecture utilizes *Layer Normalization* coupled with *Residual Connections*, also known as *Skip Connections*. As data flows through deep neural networks, the continuous updates to the network's weights during training cause the distribution of the mathematical values to fluctuate wildly. This makes it harder for subsequent layers to adapt and learn, a phenomenon known as internal covariate shift. Layer Normalization solves this by re-balancing the numbers inside each individual token's vector independently. It adjusts these values so that their average is continually reset to zero and their spread is kept tightly controlled, with a standard deviation of one, ensuring a clean and consistent signal for the next layer.

In the standard architecture, this normalization is always applied immediately after a Residual Connection that entirely bypasses the main operation [19]. The exact formulation used by the authors to describe the final output of each internal block, whether it is an attention block or a FFN, is:

$$\text{Output} = \text{LayerNorm}(x + \text{Sublayer}(x)) \tag{2.8}$$

In this equation, x represents the original input data flowing into the block, while $\text{Sublayer}(x)$ represents the new data after it has been processed by the attention or feed-forward mechanism. The addition operator (+) constitutes the Residual Connection: it simply adds the unprocessed original input back to the processed output. This mathematical design ensures that original information is never lost and helps error signals flow smoothly backward during training without vanishing. Finally, the LayerNorm function wraps this combined result, normalizing it before passing it to the next stage.

Following the attention blocks, the data passes through a **Position-wise Feed-Forward Network (FFN)** [19]. While the attention mechanism aggregates information and captures relationships across different tokens, the FFN applies learned transformations to each position in the input independently and identically. Its purpose is to add depth and non-linearity to the network. As described in the foundational architecture, this is typically implemented as two **Multi-Layer Perceptron (MLP)** linear transformations with a **Rectified Linear Unit (ReLU)** activation function in between:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{2.9}$$

The hidden space within this FFN often projects the data into a much higher dimensionality (e.g. from $d_{model} = 512$ to $d_{ff} = 2048$) before bringing it back down, allowing the model to approximate greater complexity [19].

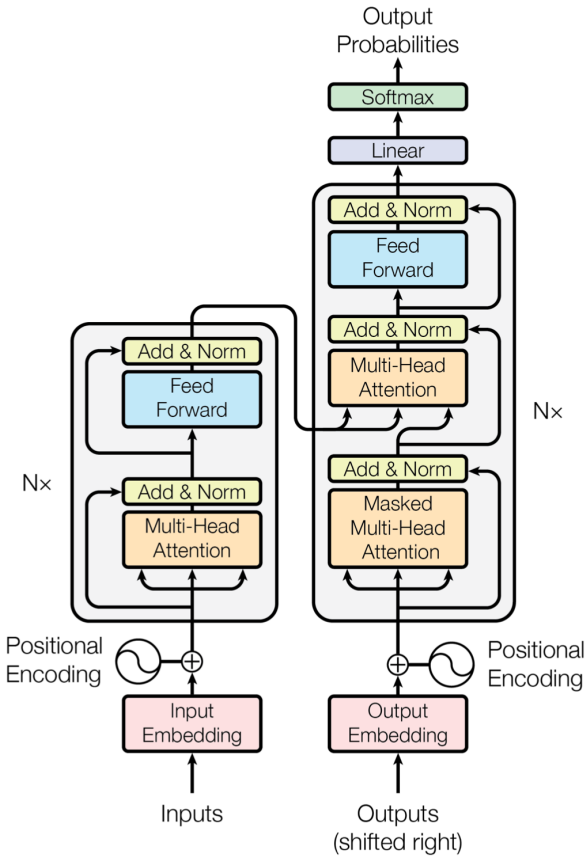

Figure 2.5: The Transformer encoder-decoder architecture and the flow of source and target sequences [19].

Putting all these components together results in the complete Encoder-Decoder architecture. The encoder takes the source sequence, processes it through multiple identical layers of self-attention and FFNs, and passes its final contextualized representations to the decoder. The decoder takes the target sequence, applies masked self-attention, and then uses cross-attention to query the encoder's representations before making a final prediction via a linear projection layer and a softmax function [19].

While the original Encoder-Decoder paradigm is extremely powerful for sequence transduction and translation, modern LLMs have predominantly shifted towards *Decoder-Only* architectures. In a Decoder-Only model, the encoder block is entirely discarded. The model processes the input prompt and generates the output using solely stacked layers of masked self-attention and FFNs. This architectural shift relies on the premise that a sufficiently deep and wide decoder, trained on massive amounts of data using *Causal Language Modeling (CLM)*, can implicitly learn to encode the prompt within its initial hidden states and seamlessly transition into generating the requested output. This significantly reduces the overall parameter complexity and optimizes the memory allocation on GPUs during inference.

2.1.5 Positional Encoding and Context Window

As can be seen from the mathematical definition of the scaled dot-product attention (Equation Self-Attention and Multi-Head Attention), the attention operator is inherently position-invariant. Because the model contains no recurrence and no convolution, it treats the input sequence as a "*bag of words*" [19]. If we were to completely shuffle the input tokens, the attention mechanism would output the exact same results, just shuffled accordingly. For the purpose of sequence modeling and language generation, this represents a problem: changing the order of words alters the semantic meaning of a sentence, and the model must be aware of this sequential nature.

To address this, there is a strict requirement to make the output of the attention mechanism dependent on the absolute and relative order of the input tokens. This is accomplished by injecting **Positional Encodings (PEs)** into the input embeddings at the bottom of the encoder and decoder stacks, before they are ever processed by the attention blocks [19].

Historically, there are two primary approaches to this problem. The classical implementation, as proposed in the original architecture, utilizes fixed sinusoidal functions, such as sine and cosine

waves of varying frequencies, to create a unique positional signature for every index in the sequence. Specifically, the PE for a position pos and dimension i is mathematically defined as:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{2.10}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{2.11}$$

This specific mathematical formulation allows the model to easily learn to attend by relative positions of the input tokens. For any fixed offset k, the Positional Encoding PE_{pos+k} can be represented as a linear function of PE_{pos} [19].

The other widely adopted solution consists of learnable PEs. In this approach, a learnable tensor of the exact same shape as the input embedding is initialized, and during the training phase, the model inherently learns the optimal vector representations for each position. Regardless of the method chosen, these positional vectors are constrained to have the exact same dimensionality d_{model} as the token embeddings: this allows them to be simply added element-wise ($Embedding + PE$), providing the network with a geometric sense of sequence order without altering the structural tensor shapes [19].

This architectural choice, however, comes with a significant trade-off directly related to the **Context Window**. While the attention mechanism allows for an infinite receptive field and high parallelization, representing a massive advantage over RNNs, its main drawback lies in the computational complexity of the attention matrix itself. Computing the similarity between every single token and every other token means that the number of operations and the memory footprint grow quadratically with the length of the input sequence, mathematically expressed as $O(n^2)$ [19].

Consequently, standard Transformers are notoriously challenging to use for managing very long inputs. If one attempts to feed a massive, extensive document into the model, the memory allocation inside the GPU VRAM grows dramatically, quickly leading to **Out Of Memory (OOM)** errors. This bottleneck defines the model's Context Window, the maximum number of sequential tokens it can successfully process at once. Despite this limitation, which is heavily researched and mitigated through various advanced architectures and sparsity techniques in modern LLM, the Transformer has imposed its dominance on the world of AI, rendering almost any other architecture devoted to sequence processing obsolete.

2.1.6 ViT: Vision Transformer

While the Transformer architecture has established itself as the de-facto standard for NLP, its application to *Computer Vision* was traditionally limited by the dominance of CNNs. These networks rely on inductive biases, such as translational invariance and a local receptive field, to process images. The **Vision Transformer (ViT)** was introduced to challenge this paradigm, demonstrating that a pure Transformer architecture can achieve state-of-the-art results in image recognition by treating images as sequences of patches, effectively processing them like sentences [6].

The core logic of ViT lies in its unique tokenization mechanism. Since a Transformer requires a one-dimensional sequence of embeddings as input, a two-dimensional image must be reshaped. To achieve this, the model divides the original image into fixed-size square *patches* (e.g. 16×16 pixels). Each patch is then flattened into a vector and mapped through a learnable linear projection to a constant latent vector size d_{model}. This process can effectively transform an image into a sequence of "visual words".

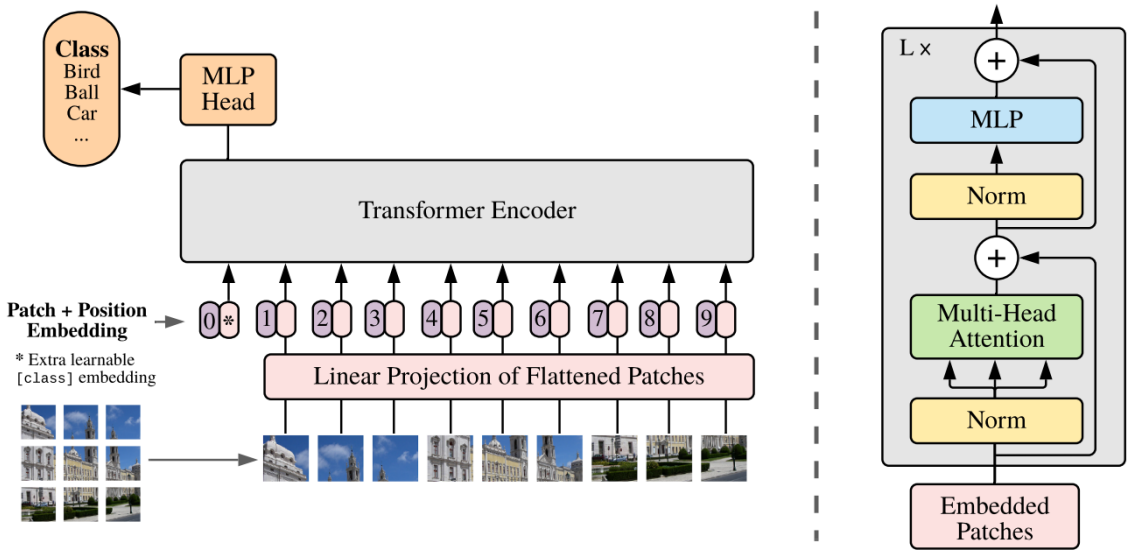

Figure 2.6: Architecture of the ViT, illustrating how image patches are linearly projected and processed by a standard Transformer Encoder [6].

The internal mechanism of ViT is almost identical to the standard Transformer encoder described in the previous sections, with two specific adaptations:

- **Class Token:** Similar to the *Bidirectional Encoder Representations from Transformers (BERT)* model in NLP, a learnable [*class*] token is prepended to the sequence of *patch embeddings*. The state of this token at the output of the Transformer serves as the global representation of the image, which is then passed to a classification head.

- **Position Embeddings:** Since the attention operator is position-invariant, the model must be informed about the spatial arrangement of the patches. Learnable one-dimensional position embeddings are added to the patch embeddings to retain spatial information.

The main difference between ViT and a standard NLP Transformer is the nature of the input data: while the latter processes discrete tokens from a vocabulary, ViT processes continuous projections of raw image pixels. Unlike CNNs, which have a hard-coded local search space, ViT utilizes the global self-attention mechanism to capture relationships between any two patches across the entire image from the very first layer [6].

The utility of ViT is particularly evident when they are pre-trained on massive datasets, such as ImageNet-21k or JFT-300M. In these scenarios, ViT outperforms traditional CNNs by exhibiting a superior ability to scale, achieving higher accuracy on classification tasks while requiring significantly fewer computational resources for training. This architectural shift has paved the way for a unified approach to AI, where the same Transformer-based principles can be applied to both text and vision with minimal modifications.

2.2 Training

The training of LLMs is an extensive and multi-stage process designed to transform a randomly initialized neural network into an intelligent system capable of understanding and generating human-like text. This process is primarily divided into two main phases: **Pre-training** and **Fine-tuning**, each serving a distinct purpose in the model's development [18, 2]. Running this process, especially for LLMs, requires substantial computational resources and a suitable infrastructure that can handle the consumption of many GPU hours while updating the model weights [18].

The first stage is **Pre-training**, which involves exposing the model to a massive corpus of text in a self-supervised manner. Modern models can be trained even on enormous scales; for instance,

the Llama 2 family was trained on a corpus of two trillion tokens from publicly available sources, with a focus on data quality and diversity [18]. To ensure the model learns robust information, these pre-training corpora are filtered to remove personal data and upsample reliable, factual sources.

During this phase, the model learns the fundamental statistical and semantic structures of language by predicting the next token in a sequence. Through millions of iterations, the network's parameters are mathematically adjusted to minimize the prediction error. This simple yet powerful objective allows the network to capture syntax, world knowledge, and reasoning capabilities implicitly, storing them directly within its weights.

A defining characteristic of this phase is the *Scaling Law*: as the number of parameters and the volume of training data increase, the model's performance improves predictably. As demonstrated by GPT-3, which features one hundred and seventy-five billion parameters, such scale allows the model to become a few-shot learner, capable of performing new tasks by processing only a few examples provided in the prompt, without requiring any gradient updates or further training [2]. This effectively overcomes the traditional paradigm that strictly required thousands of task-specific fine-tuning examples [2].

Following pre-training, the resulting base model possesses vast knowledge but lacks alignment with human intent, safety constraints, or specific conversational styles. To bridge this gap, the model undergoes **Fine-tuning**. This stage often begins with *Supervised Fine-tuning (SFT)*, where the model is trained on high-quality datasets of human-annotated instructions to learn how to follow prompts and engage in dialogue. In the development of Llama 2 - Chat, over one million human-annotated examples were utilized to optimize the model for helpfulness and safety [18].

To further refine the model's behavior, modern architectures can employ *Reinforcement Learning with Human Feedback (RLHF)*. This mechanism consists of an iterative process that involves collecting human preferences to train a reward model that can distinguish between LLM responses, rating them as better or worse. The LLM is then optimized using techniques such as *Rejection Sampling* and *Proximal Policy Optimization (PPO)* to maximize its performance based on reward model ratings [18]. Furthermore, Safety Tuning is integrated throughout the entire RLHF process. This involves red teaming, where human experts proactively simulate adversarial attacks to identify model vulnerabilities, and the use of safety-specific datasets designed to teach the network how to appropriately refuse dangerous or unethical requests [18].

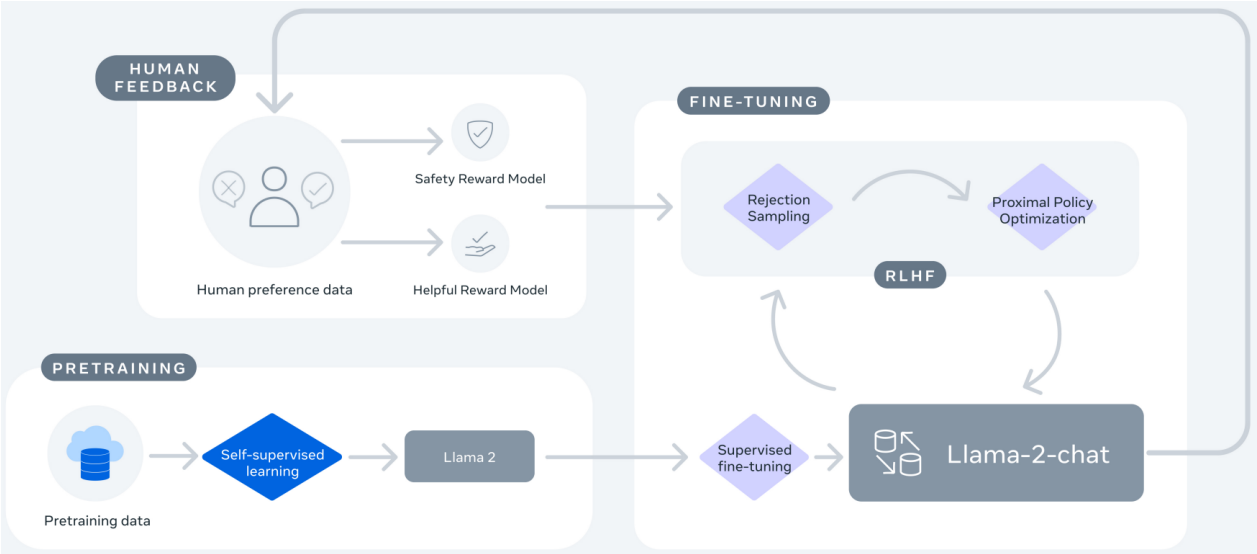

Figure 2.7: Training pipeline of Llama-2-chat, detailing the progression from self-supervised pretraining to SFT and iterative RLHF [18].

This comprehensive pipeline ensures that the final model is not merely a passive repository of knowledge, but a safe assistant strictly aligned with human values and practical utility.

2.2.1 Causal Language Modeling vs Masked Language Modeling

The effectiveness of the pre-training phase relies entirely on the specific learning task the model is forced to solve. Because the network is trained on vast amounts of raw, unlabeled text, it requires a self-supervised mechanism to continually test and improve its own knowledge. In the landscape of modern LLMs, two dominant training paradigms have emerged to serve this purpose: **Masked Language Modeling (MLM)**, which functions as a complex "*fill-in-the-blank*" exercise, and **Causal Language Modeling (CLM)**, which operates as a continuous next-word prediction engine. These two distinct approaches fundamentally define how the network sees the text and, consequently, what kind of linguistic intelligence it develops.

MLM, brought to the forefront by the BERT architecture, operates on a bidirectional principle [5]. During training, the model is presented with a sentence where a certain percentage of tokens, typically 15%, are randomly replaced with a specific [*MASK*] token. The objective of the network is to predict the original identity of these hidden tokens by looking at the entire surrounding context, both to the left and to the right of the mask [5]. This denoising task forces the model to develop

a deep, holistic understanding of how words relate to each other within a sentence. Because it can see the future and the past of a sequence simultaneously, MLM is exceptionally powerful for discriminative tasks such as sentiment analysis, named entity recognition, or sentence classification, where understanding the full context is more important than generating new text.

In contrast, **CLM** is an autoregressive approach that serves as the foundation for generative models like GPT-3 [2]. Unlike the bidirectional nature of MLM, CLM is unidirectional: the model is trained to predict the next token in a sequence given only the preceding tokens. This means that at any given point during training, the network is mathematically prohibited from looking ahead at the tokens it is trying to predict [2]. This constraint is what allows CLM models to be so efficient at generating text. By learning the probability distribution of what word most likely follows another, the model can, for example, extend the concepts of a prompt while remaining consistent with the context of the conversation.

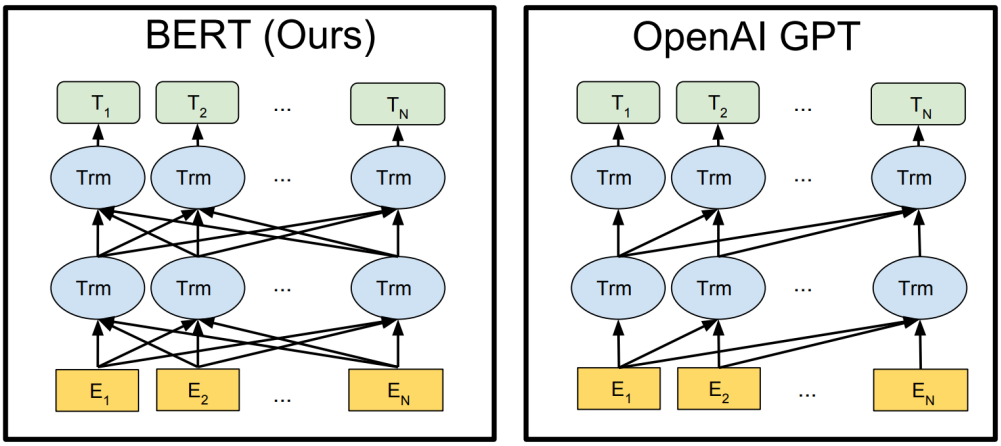

Figure 2.8: Comparison of pre-training architectures illustrating BERT's bidirectional Transformer and OpenAI GPT's unidirectional approach [5].

The choice between these two paradigms involves a trade-off in representation capability. While MLM produces richer, more contextual embeddings by considering the full sentence structure, it is difficult to use for fluid text generation because it never learns to predict sequences step by step. On the other hand, while CLM might have a slightly more limited view of the context at any single moment, its ability to scale and perform few-shot learning has made it the preferred choice for the modern generation of LLMs [2]. By excelling at the task of predicting the next token over an ever-increasing number of examples, CLM models develop an emergent ability to reason, reason,

summarize, and even write code, demonstrating that a unidirectional generative goal can lead to more complex general intelligence.

2.2.2 Supervised Fine-Tuning and Instruction Tuning

The transition from a pre-trained base model to a specialized model capable of executing specific tasks is governed by the paradigm of SFT and its more specialized evolution, **Instruction Tuning**. While pre-training on massive datasets allows models to acquire a vast internal representation of language through next-token prediction, it does not inherently teach the model how to adhere to specific user intents or follow structured commands. SFT is the general process of further training a pre-trained model on a labeled dataset where the target output is already known [23].

However, within this broad category, Instruction Tuning has emerged as an evolution of SFT capable of aligning models with human requirements. Instruction tuning refers specifically to the process of fine-tuning LLMs on a collection of datasets formatted as (instruction, output) pairs. This methodology addresses the gap between the statistical objective of predicting the next word and the human objective of having a model that effectively responds to natural language prompts.

The logic behind these techniques exists because foundational models often struggle to adapt to customized tasks without explicit guidance. While SFT can be used for general task adaptation, Instruction Tuning is preferred when the goal is to enhance the model's controllability and its ability to generalize across unseen instructions [23]. The primary difference lies in the data structure: while standard SFT might simply involve text completion on a specific domain, Instruction Tuning typically utilizes a more complex triplet format consisting of an instruction, an optional input which serves as context, and the desired output that serves as ground truth. By training on thousands of these triplets, the model does not just learn new information; it learns the instruction-following behavior itself. This allows the model to achieve superior performance in *zero-shot* scenarios, where it can interpret and execute a new task simply by understanding the linguistic structure of the prompt, rather than relying solely on the patterns seen during the initial pre-training [23].

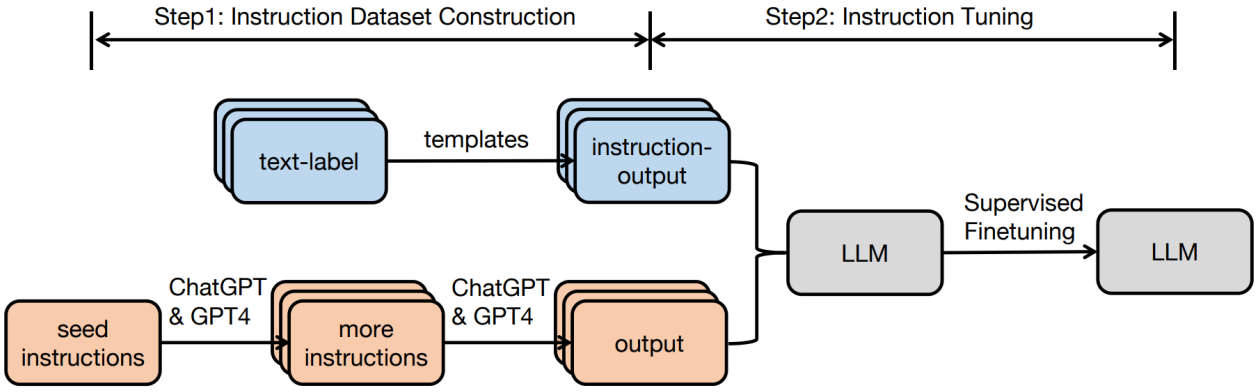

Figure 2.9: General pipeline of instruction tuning [23].

An important component that ensures the success of this alignment is **Instruction Formatting**. This concept refers to the systematic organization of the raw data into a structured template that the model can interpret during training and inference. Instruction formatting serves a dual purpose: it provides a clear boundary between the task description, for example by defining the "system" or "user" prompt, and the data to be processed, while also defining the expected structural response. The choice of format is important for the model's robustness and controllability [7].

Modern implementations often move beyond simple text-to-text formatting, which is typically referred to as **TextTuning**. Existing TextTuning methods frequently struggle with issues such as generalization, robustness, and controllability due to their lack of explicit task structures [7]. To overcome these limitations, modern implementations move toward more complex structures, such as using specific roles, e.g., system, user, and assistant; or structured schemas like JSON. This structured approach, known as *structure-to-structure tuning*, is the foundation of **JsonTuning**, a paradigm that uses JSON structures to explicitly represent tasks. JsonTuning improves generalization by clarifying task elements and their relations, boosts robustness by minimizing ambiguity, and enhances controllability by allowing precise control over outputs [7]. By using a predefined format, constraints can be applied to the output, ensuring that the model acts as a reliable data extractor that follows a specific logical flow rather than generating a conversational or rambling response.

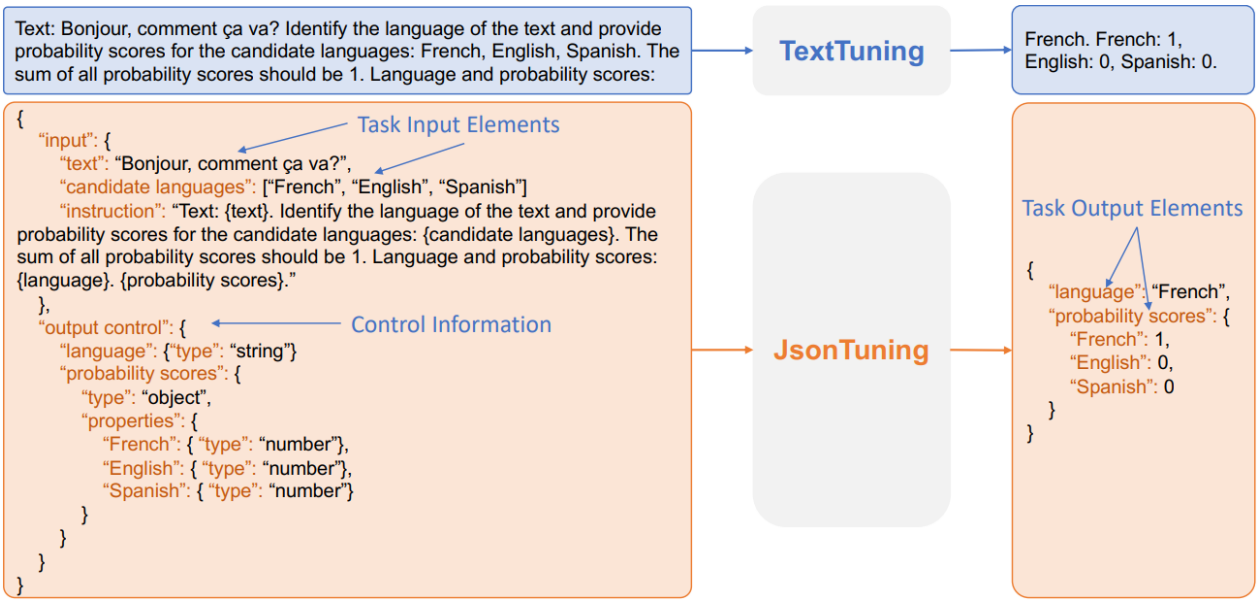

Figure 2.10: Overview of the typical TextTuning method and our proposed JsonTuning paradigm [7].

In summary, while SFT provides the general framework for supervised learning, Instruction Tuning and the accompanying Instruction Formatting increase the model's accuracy and enforce the output's structural format for that specific task.

2.2.3 Natural Language Processing: Evaluation Metrics

The evaluation of natural language generation systems represents one of the most complex and at the same time useful challenges in AI. Unlike objective tasks where a single correct answer exists, language is subjective, and a single prompt can result in multiple valid outputs. Traditionally, human evaluation has been considered the gold standard; however, it is notoriously slow, expensive, and difficult to scale. To bridge this gap, automatic evaluation metrics are employed to provide a fast and reproducible way to measure the quality of the text generated by a model [3]. These metrics generally aim to quantify the similarity between the model's output, called the candidate, and one or more human-written references, focusing on aspects such as fluency, adequacy, and semantic coherence.

In the evaluation of natural language, an *n-gram* is defined as a contiguous sequence of n items, typically words, extracted from a given text. The most widely utilized family of metrics is based on

n-gram overlap, which calculates the precision or recall of words shared between the candidate and the reference. The **Bilingual Evaluation Understudy (BLEU)** is the foundational metric in this category, primarily measuring n-gram precision. It computes how many n-grams in the generated text appear in the reference, modified by a brevity penalty to prevent the model from favoring unnaturally short outputs [3].

The overall score for BLEU is mathematically defined as:

$$\mathrm{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases} \tag{2.12}$$

$$\mathrm{BLEU} = \mathrm{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{2.13}$$

Here, based on the original formulation [13], there are several parameters. Specifically, c represents the total word length of the candidate translation generated by the model, while r is the effective length of the human reference corpus. The term BP, which stands for *Brevity Penalty*, is introduced as a safeguard: it explicitly penalizes the model if it attempts to "cheat" the metric by generating unnaturally short sentences that contain only a few highly precise words. Furthermore, p_n denotes the modified n-gram precision: the term "modified" implies that the count of matching n-grams is clipped to the maximum number of times that specific n-gram appears in the reference, preventing the model from artificially inflating its score by continuously repeating the same correct word. Finally, w_n represents the positive weights assigned to each n-gram level, typically uniform summing to one, and N is the maximum n-gram length considered in the evaluation, which is conventionally set to four in standard baseline configurations.

However, a significant limitation of standard BLEU is that the score drops to zero if a higher-order n-gram, such as a 4-gram, does not match, which is common in short sequences. To mitigate this, **BLEU with smoothing** is often implemented: this technique adds a small positive value to zero n-gram counts, ensuring that the metric provides a meaningful score even when matches are imperfect.

In contrast to the precision-oriented focus of BLEU, the **Recall-Oriented Understudy for Gisting Evaluation (ROUGE)** family focuses on recall, measuring how much of the human reference is captured by the model. This is especially relevant for tasks where all key information

needs to be covered. A variation of the classic ROUGE is **ROUGE Longest Common Subsequence (ROUGE-L)**, which relies on the longest common subsequence between the candidate and the reference. By identifying the longest sequence of words that appear in both texts in the same relative order, ROUGE-L can capture sentence structure more effectively than simple n-gram matches, as it does not require consecutive sequences [3].

The ROUGE-L metric computes precision, recall, and a final harmonic mean defined as:

$$R_{lcs} = \frac{LCS(X,Y)}{m} \quad \text{and} \quad P_{lcs} = \frac{LCS(X,Y)}{n} \tag{2.14}$$

$$F_{lcs} = \frac{(1+\beta^2)R_{lcs}P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}} \tag{2.15}$$

In these formulas, $LCS(X,Y)$ indicates the length of a longest common subsequence of X and Y, where X is a reference summary sentence of length m and Y is a candidate summary sentence of length n. R_{lcs} computes the recall, P_{lcs} calculates the precision, and F_{lcs} represents their weighted harmonic mean, where $\beta = P_{lcs}/R_{lcs}$ [12].

Another variant for NLP metrics is the **Metric for Evaluation of Translation with Explicit Ordering (METEOR)**. This metric improves upon BLEU by incorporating stemming and synonymy matching, meaning it can recognize that two words are related even if they are not identical. METEOR calculates a score based on the harmonic mean of unigram precision and recall, with a specific penalty for fragmented word order. This allows it to correlate more closely with human judgment regarding the fluency and meaning of a sentence [3].

The final METEOR score is formulated as:

$$F_{mean} = \frac{10PR}{R + 9P} \tag{2.16}$$

$$Pen = 0.5\left(\frac{c}{u_m}\right)^3 \tag{2.17}$$

$$\text{Score} = F_{mean} \cdot (1 - Pen) \tag{2.18}$$

Here, P and R represent the unigram precision and unigram recall. The term Pen stands for the fragmentation penalty, where c is the number of chunks (contiguous and identically ordered mapped words) and u_m is the number of unigrams that have been mapped [1].

Finally, for tasks requiring a consensus-based approach among multiple references, the **Consensus-based Image Description Evaluation (CIDEr)** can be utilized. Originally designed

for image captioning, CIDEr applies a *Term Frequency-Inverse Document Frequency (TF-IDF)* weighting to n-grams, giving more importance to rare, highly informative words while discounting common, less meaningful ones [20].

The CIDEr score for a specific n-gram length n is calculated as:

$$\text{CIDEr}_n(c_i, S_i) = \frac{1}{m} \sum_{j=1}^{m} \frac{\boldsymbol{g}^n(c_i) \cdot \boldsymbol{g}^n(s_{ij})}{\|\boldsymbol{g}^n(c_i)\| \|\boldsymbol{g}^n(s_{ij})\|} \tag{2.19}$$

In this equation, c_i is the candidate sentence and $S_i = \{s_{i1}, \ldots, s_{im}\}$ represents the set of m human reference sentences. $\boldsymbol{g}^n(c_i)$ and $\boldsymbol{g}^n(s_{ij})$ denote the TF-IDF vectors for the candidate and the j-th reference respectively. The metric calculates the average cosine similarity between these TF-IDF vectors across all references [20].

Together, these metrics form a multi-faceted framework to make evident the strengths and weaknesses of generative models.

2.2.4 Practical Limitations: Hallucinations and Computational Constraints

Despite the remarkable capabilities demonstrated by LLMs, their integration into sensitive or real-world applications is significantly hampered by the potential emergence of two categories of limitations: the phenomenon of **hallucinations** and the extreme **computational demands** required for their development. These LLM problems represent the current frontier in model optimization, as they directly impact both the reliability of the generated content and the economic feasibility of training these systems.

Hallucinations are arguably the most critical challenge regarding the reliability of LLMs. A hallucination occurs when a model generates content that is nonsensical or unfaithful to the provided source, often contradicting established world knowledge or diverging from the specific context of the user input [24].

These phenomena are generally categorized into three distinct types: **input-conflicting**, **context-conflicting** and **fact-conflicting** hallucinations [24].

Input-conflicting hallucinations occur when the generated text diverges from the specific constraints or source material provided in the user's prompt. **Context-conflicting** hallucinations arise during extended interactions, where the model generates content that contradicts its own previously generated responses, thereby breaking the logical continuity of the dialogue. Finally,

fact-conflicting hallucinations represent instances where the model confidently produces statements that misalign with established world knowledge, providing factually incorrect information despite maintaining an authoritative tone.

This behavior often stems from flaws in the pre-training data, such as noise or misinformation, and is aggravated by the standard training objectives that prioritize the statistical likelihood of tokens over their factual accuracy [24]. Consequently, the model may confidently produce plausible-sounding but entirely fabricated statements, making its output difficult to verify without expert supervision.

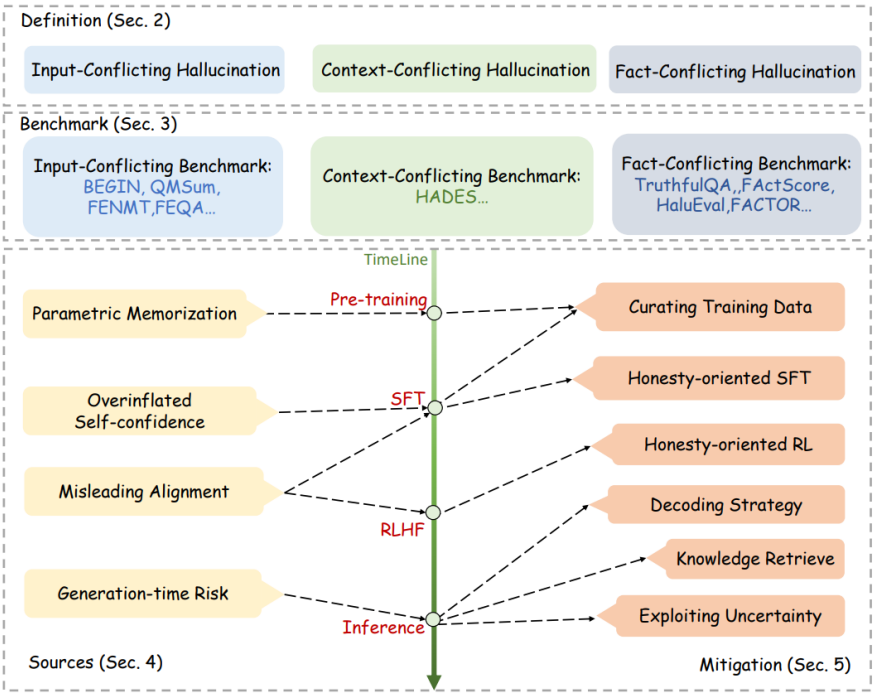

Figure 2.11: Overview of LLM hallucinations, categorizing definitions and benchmarks alongside sources and mitigation strategies mapped across the model's life cycle [24].

To mitigate this problem, several strategies can be employed throughout the model's life cycle. During the training stage, data can be curated and processed to filter out noise, or *honesty-oriented* reinforcement learning can be employed to teach the model to recognize its own knowledge boundaries and express uncertainty [24]. Alternatively, during the inference stage, hallucinations can be reduced by resorting to external knowledge bases to ground responses in verifiable facts, analyzing the model's uncertainty to detect fabrications, or leveraging multi-agent interactions where different

models collaboratively debate to reach a factual consensus [24].

Parallel to these reliability issues are the rigid computational constraints governed by empirical Scaling Laws. The research conducted by Kaplan [10] highlights that the performance of a language model, measured via cross-entropy loss, follows a precise *power-law* relationship with respect to three fundamental factors: the **number of model parameters**, the **size of the dataset**, and the **total amount of compute** used for training. This implies that improving the intelligence of the model does not depend on modifying its architecture, but depends mainly on a question of scale. While larger models are significantly more sample-efficient, meaning they reach a higher level of performance with fewer training examples, they require a massive increase in hardware resources and energy consumption [10].

Furthermore, these laws suggest that to optimize a fixed compute budget, it is often more efficient to train very large models on a relatively modest amount of data and stop well before convergence, rather than training smaller models to completion. This creates many difficulties as the progress of LLMs is increasingly tied to the availability of high-end computational infrastructure, leading to a landscape where the most advanced systems are often defined by the scale of the hardware used to produce them [10]. Together, these limitations ensure that while LLMs are powerful tools, they remain systems that require careful alignment and substantial resources to be both accurate and sustainable.

2.3 Parameter Efficient Fine Tuning

The rapid growth in the parameter scale of LLMs has made the traditional approach to model adaptation increasingly unsustainable. Historically, the standard method for task-specific adaptation was FFT, which involves updating all the parameters of a pre-trained model on a downstream dataset. However, as models reach hundreds of billions of parameters, FFT becomes prohibitively expensive, requiring a huge amount of GPU memory to store the **model weights**, **gradients**, and **optimizer states** for each individual parameter [14]. To understand what this computational cost depends on, it is helpful to define these elements.

The **model weights** represent the actual parameters of the neural network, essentially the core memory that holds the learned knowledge. During the learning process, the network computes

gradients, which are mathematical vectors indicating the magnitude and direction in which each weight must be adjusted to reduce errors. Finally, the system must keep track of **optimizer states**, which are auxiliary memory variables used by optimization algorithms, such as *Adam*, to individually adjust the learning rate for each specific weight. While the model weights themselves consume only about 20% of the memory, the remaining 80% is heavily dominated by the storage of these gradients, optimizer states, and intermediate activations required during training [14].

Therefore, when attempting to specialize models on consumer-grade hardware or with limited computational budgets, the computational resource demands of the FFT make this task impractical.

PEFT emerged as a paradigm shift to address these challenges. The core objective of PEFT is to achieve performance comparable to FFT while updating only a tiny fraction of the model's parameters, often less than 1% of the total weight count [21]. This is achieved by freezing the original weights of the pre-trained backbone model and only training a small set of additional parameters, known as adapters, or selectively updating a specific subset of the existing architecture.

The primary motivations for adopting PEFT go beyond simple resource savings. By keeping the majority of the model weights frozen, PEFT effectively mitigates the problem of "*catastrophic forgetting*", a phenomenon where a model loses its general-purpose knowledge during task-specific training [14]. Furthermore, PEFToffers advantages in terms of storage and deployment efficiency. Instead of saving and distributing a full copy of the model of several hundred gigabytes for each specific task, one only needs to store the small "*delta*" weights, which often amount to only a few megabytes.

In the context of PEFT, these deltas, often mathematically denoted as ΔW, are the specific task-oriented parameter increments learned during the new training phase. Because the original knowledge of the backbone model, indicated with W, remains frozen and unmodified, the system only computes this tiny, new layer of delta information. During inference, these incremental weights are combined with the original architecture, adapting the model's behavior without permanently altering its foundational parameters [21]. This technique allows for the rapid switching of task-specific behaviors within a single deployed model backbone [21].

Over the years, the PEFT environment has matured, and several methodologies have been developed to implement these efficiency improvements, ranging from adding layers to reparameterizing existing weight matrices. In the following subsections, we will explore the technical distinctions

between these approaches and focus on the specific techniques most widely used.

2.3.1 Differences between Full Fine Tuning and Parameter Efficient Fine Tuning

The distinction between **FFT** and **PEFT** lies in the strategy used to modify the internal knowledge of a model for a specific task. While both approaches aim to adapt a pre-trained model to a downstream application, they operate under different constraints regarding **computational overhead**, **memory management**, and **information preservation**.

The most evident difference is the parameter update strategy. In FFT, the training process treats the entire architecture as a set of trainable parameters. This means that every single weight in the model is subject to adjustment via backpropagation. In contrast, PEFT techniques freeze the vast majority of the pre-trained weights, focusing the training process on a minimal subset of parameters. This subset often represents less than 1% of the original model size, yet it is capable of capturing the necessary task-specific nuances [21].

This difference leads to a massive disparity in computational and storage costs. FFT requires storing the gradients and optimizer states for the entire model, which typically demands three to four times the GPU memory compared to the model's static weight size. For a modern LLM, this often necessitates the use of high-end industrial clusters. PEFTdrastically reduces this footprint because gradients and optimizer states are only calculated and stored for the small percentage of trainable parameters. Furthermore, from a storage perspective, FFT produces a complete, multi-gigabyte copy of the model for every new task. PEFT, however, generates only a small checkpoint or adapter file, weighing a few megabytes, allowing a single foundation model to support specialized tasks by simply swapping these lightweight deltas [14].

Another differentiation is the risk of *catastrophic forgetting*. Because FFT modifies all original weights, it frequently overwrites the general-purpose knowledge the model acquired during pre-training, potentially degrading its performance on tasks outside the fine-tuning domain. PEFT protects this foundational knowledge by keeping the backbone weights frozen. This preservation ensures that the model retains its reasoning and linguistic capabilities while the specialized layers learn the new domain [14].

In terms of performance and applicability, the choice between the two methods depends on the available resources and the specific requirements of the project. FFT is often considered the performance upper bound when domain adaptation is required and computational resources are virtually unlimited. However, recent studies demonstrate that PEFT can match or even exceed FFT performance in many scenarios, as the limited number of trainable parameters acts as a form of regularization that prevents overfitting on small datasets [21]. Consequently, PEFT has become the preferred choice for modern LLM development, enabling the deployment of AI systems on more accessible and cost-effective hardware configurations.

2.3.2 Parameter Efficient Fine Tuning Techniques

Having established the fundamental differences between FFT and PEFT in the previous section, it is useful to be able to distinguish and categorize the possible PEFT techniques currently in use. The PEFT paradigm is systematically divided into three macro-families based on the specific architectural strategy employed to achieve parameter efficiency [14]. This classification arises because different downstream tasks and hardware environments demand distinct trade-offs between training speed, inference latency, and architectural complexity. Consequently, research has diverged into three different structural solutions to adapt the backbone: introducing lightweight trainable modules (**Additive methods**), selectively unfreezing isolated parts of the existing architecture (**Selective methods**), or mathematically transforming the representation of weight updates without physically altering the original model structure (**Reparameterization-based methods**) [21].

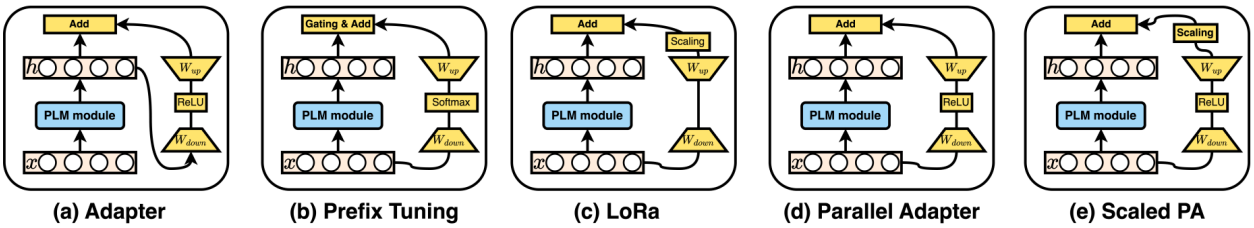

Figure 2.12: Architectural overview of key PEFT techniques, including Adapter, Prefix Tuning, LoRA, and Parallel Adapter variations [14].

Additive methods involve inserting new, small trainable modules into the existing architecture while keeping the original backbone frozen. The most representative technique in this category is the

use of *Adapters*. Historically, adapters were inserted serially between layers, typically after the FFN. An adapter usually implements a "bottleneck" architecture: it compresses the high-dimensional input into a lower-dimensional representation using a down-projection layer, applies a non-linear activation function, and then restores the original dimensionality through an up-projection layer [14]. This bottleneck forces the adapter to learn a highly compressed and efficient representation of the new task.

To further contextualize this approach, it is essential to consider the placement of these modules within a standard Transformer block. The classical serial configuration integrates the adapter sequentially, meaning it processes the intermediate representations directly after the pre-trained attention or FFN layers. By routing the data through this distinct pathway, the model isolates the learning of the new task from the general knowledge, which remains encoded in the frozen weights. While structural alternatives exist, such as parallel integration where task-specific computations occur alongside the primary layers rather than after them, the sequential typology remains the reference model of additive tuning [14].

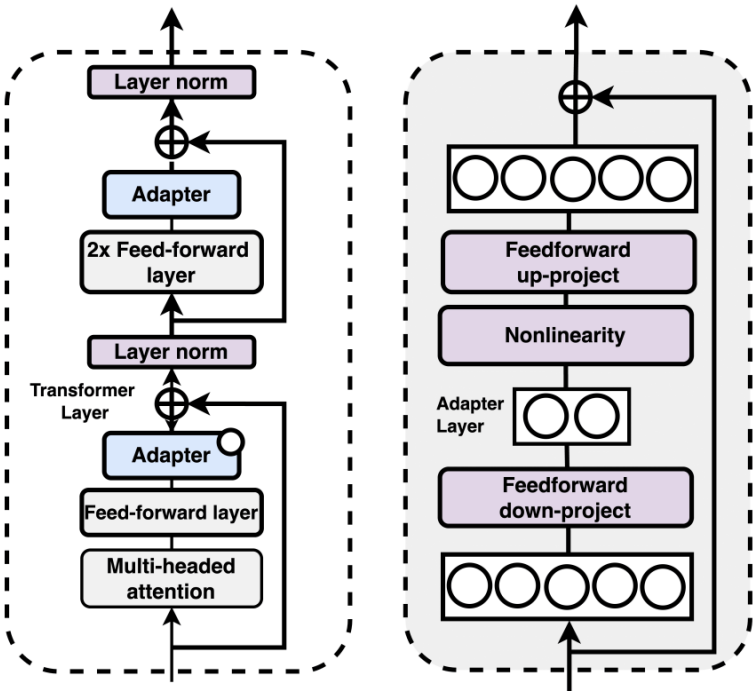

Figure 2.13: Comparison of serial adapter integration in Transformer architecture (left) and adapter layer structure (right) [14].

Another significant additive technique is *Prefix Tuning*. Instead of adding new neural layers,

Prefix Tuning prepends a sequence of learnable continuous vectors, known as prefixes, directly to the keys and values of the multi-head attention blocks at every layer of the Transformer. These prefixes act as task-specific tokens that guide the model's attention mechanism [21]. Because they are integrated deeply across all layers rather than just at the input embedding level, they offer substantial expressive power, allowing the model to learn complex task-specific contexts while the majority of original parameters remain untouched and frozen [21].

Selective methods, or *subset-based fine-tuning*, take a fundamentally different approach. Rather than adding new modules, they identify and update only a specific small fraction of the existing pre-trained weights. A prominent strategy in this category is BitFit, which exclusively updates the bias terms of the network while freezing all other weights. Despite bias terms constituting a minuscule percentage of the total parameters, tuning them has proven effective for many downstream tasks [14]. Another common selective approach involves layer freezing, where updates are restricted primarily to the top-most layers of the model, which typically capture task-specific semantics, leaving the linguistic structures in the earlier layers intact. A major architectural advantage of selective methods is that, by not introducing any additional structural modules, they completely avoid the extra inference latency that often accompanies additive techniques like adapters [21].

The third family consists of **Reparameterization-based methods**, which leverage mathematical transformations to modify how the model's weight updates are represented. The most prominent and widely adopted technique in this category is *Low-Rank Adaptation (LoRA)*. LoRA is built on the hypothesis that the complex changes in weights required during task adaptation actually reside in a low "intrinsic dimension" [9]. Therefore, instead of updating the massive original weight matrices, LoRA freezes them and approximates the necessary updates using the product of two small, low-rank matrices. This drastically reduces the number of trainable parameters while maintaining high expressive power.

To push this efficiency even further for extremely large models, *Quantized Low-Rank Adaptation (QLoRA)* was introduced. QLoRA enhances the standard LoRA framework by quantizing the frozen pre-trained weights of the base model into a 4-bit data type known as NormalFloat4 (NF4). This is coupled with techniques like Double Quantization and Paged Optimizers to manage memory spikes during the learning process [4]. This setup enables the fine-tuning of massive models on standard hardware with significantly limited VRAM.

Because LoRA and QLoRA represent the core training methodologies utilized and compared in this thesis project, their specific mathematical properties and architectural behaviors will be analyzed in profound detail in the subsequent dedicated subsections.

2.3.3 LoRA: Low-Rank Adaptation

The most significant advancement in the field of PEFTis represented by LoRA. This technique addresses the inefficiency of FFT by leveraging a hypothesis regarding the nature of neural network adaptation.

The development of LoRA is rooted in an observation from earlier research: over-parametrized models actually reside on a low intrinsic dimension. Specifically, even though the total parameter space of a model is massive, the change in weights required to adapt the model to a new task can be captured in a much smaller subspace. Recent research has taken this concept a step further by hypothesizing that weight updates during adaptation also have low intrinsic rank [9].

We now define the mathematical parameters involved in the adaptation process. Let us consider a single pre-trained weight matrix within the model, denoted as $W_0 \in \mathbb{R}^{d \times k}$. In this notation, d represents the output dimension and k represents the input dimension of the layer, meaning the matrix contains a total of $d \times k$ individual parameters. During FFT, the training process updates this entire matrix by calculating a full-size adjustment matrix, called the ΔW. The final, adapted knowledge of the layer becomes the sum of the original weights and this update: $W = W_0 + \Delta W$. Because ΔW has the exact same dimensions as W_0, FFT forces the system to learn and store an immense number of new parameters ($d \times k$) for every single layer.

The *low-rank hypothesis* changes this approach. It suggests that the adjustment matrix ΔW does not need to be a full-rank matrix, meaning it does not need all its degrees of freedom to learn the new task. Instead, the necessary updates can be constrained to a much lower rank, denoted as r. This rank r is a hyperparameter representing a structural bottleneck, chosen to be significantly smaller than both d and k, as $r \ll \min(d, k)$. By forcing the update to occur within this low-rank constraint, LoRA approximates the massive ΔW matrix by decomposing it into two much smaller matrices. Consequently, rather than updating $d \times k$ parameters, LoRA only needs to train $(d \times r) + (r \times k)$ parameters. This mathematical decomposition effectively decreases the computational burden by several orders of magnitude without sacrificing the model's expressive power.

Thus, for a pre-trained weight matrix $W_0 \in \mathbb{R}^{d \times k}$, the forward pass is modified as a product of small, trainable matrices:

$$h = W_0 x + \Delta W x = W_0 x + BAx \tag{2.20}$$

Where:

- $x \in \mathbb{R}^k$ is the input vector.
- $W_0 \in \mathbb{R}^{d \times k}$ is the pre-trained weight matrix, which remains frozen during training.
- $\Delta W \in \mathbb{R}^{d \times k}$ represents the task-specific weight update matrix.
- $A \in \mathbb{R}^{r \times k}$ and $B \in \mathbb{R}^{d \times r}$ are the low-rank decomposition matrices that contain the only trainable parameters.

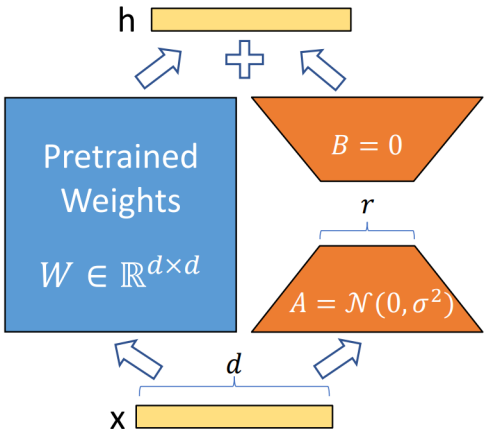

Figure 2.14: LoRA reparametrization, illustrating frozen pretrained weights alongside the trainable low-rank matrices A and B [9].

During training, W_0 receives no gradient updates: the system only optimizes the matrices A and B. It is interesting to note that the rank r is a hyperparameter that determines the capacity of the model's adaptation. Empirical evidence shows that even a very small rank, such as $r = 1, 2, 4$, or 8, is often sufficient to achieve performance comparable to FFT, even for models as large as GPT-3 with one hundred and seventy-five billion parameters [9].

To ensure that the training process starts effectively and that the initial behavior of the model is identical to the pre-trained version, LoRA utilizes an initialization strategy. The matrix A

is initialized using a random Gaussian distribution, while the matrix B is initialized to zero. Consequently, at the beginning of the fine-tuning process, the product $\Delta W = BA$ is zero:

$$\Delta W = B \cdot A = 0 \cdot A = 0 \tag{2.21}$$

This ensures that the initial output h is exactly $W_0 x$, meaning the model starts the adaptation process from the original pre-trained baseline without any random noise being injected into the hidden states.

Furthermore, LoRA applies a scaling factor to the low-rank update. The term BAx is scaled by a factor of $\frac{\alpha}{r}$:

$$h = W_0 x + \frac{\alpha}{r} BAx \tag{2.22}$$

In this equation, α is a constant hyperparameter. When optimizing with Adam or similar optimizers, tuning α is the same as tuning the learning rate if the initialization is scaled appropriately [9]. A common practice is to set α to the value of the first rank r chosen and then keep it constant. This scaling reduces the need to retune the learning rate when the rank r is varied, simplifying the hyperparameter search process [9].

In the Transformer architecture, LoRA is primarily applied to the weight matrices in the self-attention modules. Specifically, the study focuses on the **Query** (W_q), **Key** (W_k), **Value** (W_v), and **Output** (W_o) matrices. While LoRA can be applied to the MLP modules as well, applying it to the attention weights, particularly W_q and W_v, has been shown to yield the best balance between efficiency and performance.

An investigation into rank-deficiency reveals that the subspace spanned by the learned low-rank matrices A and B often overlaps across different ranks. This suggests that the adaptation primarily focuses on a few specific directions in the feature space [9].

One of the most powerful features of LoRA is the ability to eliminate inference latency. Unlike serial adapter layers that add depth to the model, the low-rank matrices in LoRA can be mathematically merged with the pre-trained weights once training is complete. Because both W_0 and BA are matrices of the same dimension, they can be summed into a single matrix W_{merged}:

$$W_{merged} = W_0 + BA \tag{2.23}$$

When the model is deployed, it uses W_{merged} for computation, meaning the inference speed is identical to the original pre-trained model. Furthermore, if the model needs to be adapted for a

different task, the original W_0 can be restored by simply subtracting the task-specific BA, allowing for rapid and lightweight task switching without storing multiple full copies of the model backbone.

2.3.4 QLoRA: Quantized Low-Rank Adaptation

Building upon the foundations of LoRA, QLoRA represents a secondary evolution in the democratization of LLM training. While LoRA successfully reduced the number of trainable parameters by introducing low-rank matrices, it still required the original pre-trained weight matrix W_0 to be loaded into memory in high-precision formats, such as 16-bit *Brain Floating Point 16 (BF16)* or *Float16 (FP16)* [4]. For massive models this meant that the frozen portion of the model could consume up to hundreds of gigabytes of of VRAM, effectively restricting fine-tuning to expensive, high-end industrial hardware. QLoRA solves this specific bottleneck by introducing a high-fidelity 4-bit quantization framework that allows backpropagating gradients through a quantized, frozen model into the low-rank adapters without losing the performance levels of full 16-bit fine-tuning [4].

The primary innovation of QLoRA lies in its ability to manage the precision trade-off through three distinct technical pillars: **NF4**, **Double Quantization**, and **Paged Optimizers** [4].

As for **NF4**, it is an information-theoretically optimal data type that builds upon the concept of *Quantile Quantization*, that is a method designed to ensure that each quantization bin contains an exactly equal number of values from the input tensor [4]. It achieves this by estimating the empirical cumulative distribution function of the input data. While highly accurate, calculating these exact empirical quantiles for the massive weight tensors of an LLM is computationally expensive. NF4 overcomes this limitation by exploiting a predictable property of neural networks: pre-trained weights almost always follow a zero-mean normal distribution [4]. Thus, instead of dynamically computing the quantiles for every block of weights, NF4 uses theoretically pre-computed quantiles based on a standard normal distribution. Because this method allocates denser quantization bins around zero, where the majority of the weights actually reside, NF4 significantly reduces the overall quantization error [4]. This mathematically tailored approach ensures that the compressed 4-bit representation retains the structural information of the original 16-bit knowledge without degradation.

The second innovation, **Double Quantization**, directly addresses the memory overhead gen-

erated by the quantization constants themselves. In standard block-wise quantization, weights are grouped into small chunks, e.g. blocks of 64 parameters, to isolate outliers, and each block is assigned a 32-bit floating-point scaling constant [4]. This adds an overhead of about 0.5 bits per parameter. Double Quantization treats these 32-bit constants as a new, secondary set of weights and quantizes them a second time down to an 8-bit format using a larger block size, typically 256. This two-step compression reduces the constant footprint from 0.5 bits to just 0.127 bits, saving approximately 0.373 bits per parameter on average [4]. While this might seem like a marginal gain at the micro-level, for a massive LLM it translates to a profound saving of several gigabytes of VRAM.

Finally, **Paged Optimizers** resolve the issue of sudden memory spikes. By leveraging NVIDIA's unified memory feature, this technique automatically allocates paged memory for the optimizer states [4]. If a VRAM spike occurs during training, which is common when processing long sequence lengths or large mini-batches, the system automatically evicts the optimizer states to the CPU RAM. Later, when the memory is required for the optimizer update step, it is paged back into the GPU. This transfer, which varies dynamically based on the operation that the system has to perform, prevents the OOM errors and allows the system to train massive models that would otherwise physically exceed the GPU's limits [4].

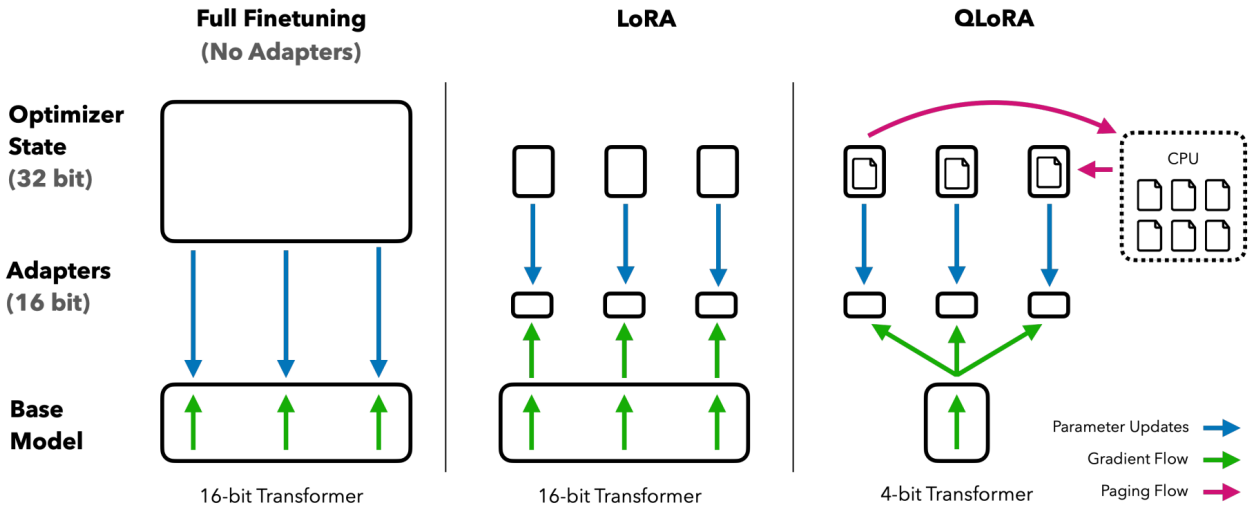

Figure 2.15: Comparison of memory requirements across FFT, LoRA, and QLoRA, highlighting QLoRA's 4-bit base model quantization and paged optimizer flow [4].

The logical flow of computation in QLoRA is optimized to maintain high precision during

the calculation of gradients. To achieve this, QLoRA separates its data types into two distinct categories: a low-precision storage data type and a higher-precision computation data type [4].

In this framework, the 16-bit BF16 format is utilized exclusively as the computation data type. Whenever a quantized 4-bit weight tensor is needed for processing, it is dequantized to BF16 to perform the forward and backward passes. This ensures that while the model's memory footprint remains small during storage, the actual matrix multiplications and the calculation of derivatives are executed in 16-bit precision, preventing performance degradation [4].

Mathematically, the interaction between the quantized base model and the LoRA adapters for a single layer is defined as:

$$Y^{\text{BF16}} = X^{\text{BF16}}\text{doubleDequant}(c_1^{\text{FP32}}, c_2^{\text{FP8}}, W^{\text{NF4}}) + X^{\text{BF16}} L_1^{\text{BF16}} L_2^{\text{BF16}} \tag{2.24}$$

To fully understand this process, the equation can be conceptually split into two distinct pathways that are summed together to produce the final layer output (Y^{BF16}).

The first part of the equation represents the **data passing through the frozen base model**. Here, X^{BF16} is the input flowing into the layer in BF16 precision. The term W^{NF4} represents the massive pre-trained weight matrix, locked in NF4 precision. The *doubleDequant* function is used to decompress: it takes the 4-bit weights and utilizes the two layers of stored quantization constants, namely c_1^{FP32} for 32-bit constants and c_2^{FP8} for 8-bit constants created by double quantization, to temporarily restore the weights back to BF16 precision for the matrix multiplication [4].

The second part of the equation ($+X^{\text{BF16}} L_1^{\text{BF16}} L_2^{\text{BF16}}$) represents the **data passing through the task-specific adapters**. L_1^{BF16} and L_2^{BF16} are the trainable low-rank adapter matrices. Unlike the base model, these adapters are never quantized; they are maintained in full BF16 precision at all times [4]. Basically, during the parameter update phase, gradients are only computed for the L_1 and L_2 adapter weights, while no weight gradients are computed for the 4-bit base model [4]. This hybrid approach ensures that the model can be stored in a compressed state, drastically reducing VRAM consumption, while the actual learning dynamics remain exactly as accurate and expressive as they would be in a standard, full-precision training environment.

Compared to standard LoRA, QLoRA offers a significant reduction in the GPU memory requirement, decreasing it by up to three times, without sacrificing task performance on evaluative benchmarks. Specifically, it maintains full 16-bit performance levels on both the *Massive Multitask*

Language Understanding (MMLU), which measures the model's factual knowledge and reasoning across diverse academic and professional domains, and the Vicuna evaluation, a framework designed to assess instruction-following capabilities and conversational chatbot quality [4]. QLoRA is therefore an excellent compromise that manages, within the context of PEFTtechniques, to shift the focus of efficiency from the quantity of hardware to the quality of the fitting process. It leverages the memory savings of NF4 and double quantization, demonstrating that even with limited computational resources, excellent results can be achieved on the largest available architectures [4].

3. Project Methodology and Implementation

Having established the theoretical foundations of LLMs, this chapter applies these concepts within a practical industrial context.

This research project stems from a specific need identified by *PMP*, a company based in Pavullo Nel Frignano, specializing in the production and processing of mechanical components. PMP operates as a contractor for *Studio ITC*, an e-commerce and mobile application development company based in the same city, where this project was carried out.

The core problem addressed is the difficulty human operators face in correctly classifying and categorizing mechanical components within PMP's management software. Understanding and processing technical drawings, both traditional and modern, presents a significant barrier for employees. A further challenge lies in training staff on the specific language and classification methods adopted internally by PMP. Technicians often have to manually review multiple documents to correctly extract the component's technical specifications, such as the exact type, required materials, and mechanical processes involved. Furthermore, manual classification is extremely time-consuming and prone to human error, especially for new hires.

Therefore, the primary objective of this thesis is to automate and optimize this workflow by developing an intelligent and scalable *pipeline* capable of autonomously extracting the most significant data from mechanical component documents. To achieve this goal, the project leverages the capabilities of modern LLMs. However, since the generic base models do not possess the specialized vocabulary and specific syntax adopted internally by PMP, it was necessary to adapt them using PEFT techniques. By doing so, the models learn to map chaotic, unstructured text into standardized data formats. This chapter serves as a comprehensive guide through the entire engineering process, detailing how the raw company data was processed to be best interpreted by the chosen models and how the latter were trained.

The architecture of the project is structurally divided into four sequential macro-phases, which also dictate the organization of this chapter.

The workflow begins with the **Data Acquisition** phase. First, the large corporate archive had to be navigated, establishing connections to the remote file system and executing filtering queries to

isolate approximately twenty thousand clean and correct mechanical documents. This was followed by parallel download strategies to ensure speed and data integrity.

After storing the data locally, the project transitioned to the **Dataset Generation** phase. Since PDF files vary greatly—ranging from digitally native documents to flat image scans—the system employed a hybrid approach combining native text extraction libraries with OCR. The extracted text, along with its corresponding *ground truth* answers provided by the corporate database, was subsequently processed, cleaned, and serialized into the JavaScript Object Notation Line (JSONL) format.

Subsequently, the focus shifts to the **Fine-Tuning Configuration** section. Training billion-parameter models requires substantial computational resources, which were provided by Unimore's HPC cluster equipped with industrial-grade GPUs. In this step, the selection of training hyperparameters and the implementation of PEFT techniques are described in detail. A direct comparison is made between standard LoRA and its quantized counterpart, QLoRA, to evaluate the real cost of mathematical compression in terms of model intelligence.

Finally, the chapter concludes with the **Inference and Evaluation** setup. To prove the effectiveness of the fine-tuning process, the models were subjected to testing on a segregated subset of unseen documents. The models' outputs were mathematically scored using standardized NLP metrics. This approach ensures that all conclusions drawn regarding model performance, scaling laws, and hardware efficiency are grounded in objective, reproducible data.

3.1 Data Acquisition

The fundamental prerequisite for adapting any AI model to a specific industrial domain is the availability of a substantial and highly representative *corpus* of training data. For this research, the source of knowledge was the **file system** maintained by PMP. This archive contains all the documents catalogued by the company since its inception, for a total of approximately seventy thousand PDF documents depicting images or cartouches of technical components. However, simply connecting and downloading the documents wasn't enough to begin the project. A closer look at the company's file system revealed that many documents were unsuitable for the project's final purpose or were corrupted. This initial acquisition phase was therefore designed to securely bridge the gap between

the company's internal storage systems and our on-premises environment, striving to create the most accurate and complete archive possible.

The first major hurdle was logical rather than physical. The corporate data was not stored in a single, easily accessible folder; rather, it was distributed across a protected remote server and managed through a *relational database*. It was essential to select and filter documents to obtain only those with specifications suitable for the project. This required the creation of a secure communication channel and the formulation of filtering instructions to query the database, ensuring that only the most relevant files were selected for extraction.

Once the theoretical list of ideal candidates was compiled, the problem shifted to the physical transfer of the files. Moving a huge volume of documents from a remote location to a local workstation presented a problem: a sequential download would have taken an unsustainable amount of time and could have saturated PMP's corporate network. Therefore, the acquisition script was designed to leverage multithreading techniques, allowing multiple virtual workers to retrieve files simultaneously. Furthermore, this phase needed to ensure the data integrity of each individual file, implementing automatic checks to prevent issues such as file corruption or name collisions during transfer.

The following subsections will delve into the specific solutions developed to orchestrate this entire acquisition pipeline. Section **Server Connection and Filtering** will explore the connection protocols and the database filtering strategies used to generate the definitive list of target files, while Section **Parallel Download and Integrity Verification** will detail the parallel download engine and the subsequent validation mechanisms employed to construct the final, clean dataset.

3.1.1 Server Connection and Filtering

The starting point of the entire data acquisition process required establishing a direct line of communication with the file system of PMP. To interface safely with the remote server where the archive is hosted, a direct connection was necessary. We established a SSH tunnel utilizing the *Paramiko* library in Python. To have a seamless, automated access procedure without the need for manual password entry during script execution, authentication was handled via *Ed25519* cryptographic keys.

In practical terms, this algorithm generates a mathematically linked pair of keys: a public key

placed on the remote server and a private key securely stored on the local machine. When the Python script attempts to connect, the server issues a mathematical challenge that can only be solved if the script possesses the correct private key. Ed25519 achieves security with significantly shorter key lengths.

This modern cryptographic standard guarantees both high security and connection speed, allowing the local downloading scripts to mount the remote file system securely and instantaneously.

Code 3.1: Establishment of the automated SSH connection

```
ssh = paramiko.SSHClient()
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

# Loading the Ed25519 cryptographic key for passwordless authentication
my_key = paramiko.Ed25519Key.from_private_key_file(KEY, password=KEY_PASSWORD)

ssh.connect(HOSTNAME, username=USERNAME, pkey=my_key)
sftp = ssh.open_sftp()
```

Once the physical connection was established, the subsequent challenge was logical: identifying exactly which documents to extract. The PMP architecture is based on a relational database that connects the data in the management software with the documents in the company file system. From the first tests, it was clear that data and document filtering was necessary before downloading, as the database contained incorrect data, corrupted documents, or references to documents that no longer exist. Therefore, we developed a specific Structured Query Language (SQL) query that acts as a data filter to prevent the extraction of corrupt, duplicate, or irrelevant files directly from the source, and to ensure that the downloaded files meet the technical specifications we want to pass to the LLM models.

The query is structurally divided into **three expressions** to systematically reduce and enrich the data.

The first logical block is designed to extract the base articles and link them to their respective PDF attachments. This block leverages **Window Function**, specifically the `COUNT() OVER (PARTITION BY...)` instruction. Unlike a standard `GROUP BY`, which compresses multiple rows into a single summary row, thus losing the individual details of each document, a Windows Function performs calculations on a specific set of related rows while keeping all the original rows intact.

In this specific query, the `PARTITION BY` clause virtually groups the data by article code or

file name, and `COUNT()` calculates the total number of occurrences within that virtual group. For example, if a single file is mistakenly linked to multiple different components in the database, the function will simply append the count value to all individual rows without deleting them. This feature calculates how many times a specific component code or a file name appears in the dataset without altering the table's structure, allowing the system to identify potential naming collisions or duplicate files directly at the database level so they can be easily filtered out in the next step.

Code 3.2: Filtering Query - First logical block: Base article selection and duplication counting

```
WITH pool_candidati AS (
    -- 1. SELECTION AND DUPLICATE CALCULATION
    SELECT
        a.id,
        a.codice,
        a.descrizione,
        a.versione,
        a.data_versione,
        elab.file AS nome_file_pdf,
        elab.estensione AS formato_file,

        -- Window Functions to identify collisions
        COUNT(*) OVER (PARTITION BY a.codice) as count_codice,
        COUNT(*) OVER (PARTITION BY elab.file) as count_file

    FROM articolo a
    INNER JOIN elaborati elab
        ON elab.id_fk1 = a.id
       AND elab.tab1 = 'articolo'
       AND elab.file ILIKE '%.pdf'
       AND elab.file NOT LIKE '%#%'

    WHERE a.codice IS NOT NULL AND btrim(a.codice) <> ''
),
```

The second block acts as the cleaning filter. It enforces uniqueness constraints by accepting only rows where both the article code count and the file count equal one. Furthermore, this section aggressively filters out a specific list of article IDs. During preliminary testing, these specific codes were found to be present in the database registry but physically missing from the company's file system, which would have caused errors during the download phase. It is within this block that the limit of twenty thousand total documents is applied, ensuring the dataset size remains consistent for

the training phase.

Code 3.3: Filtering Query - Second logical block: Filtering unique files and handling exceptions

```
articoli_puliti AS (
    -- 2. CLEANING FILTER AND EXCLUSION OF MISSING FILES
    SELECT *
    FROM pool_candidati
    WHERE count_codice = 1  -- Only unique codes
      AND count_file = 1    -- Only unique files

      AND codice NOT IN (
        -- IDs of articles that do not have a physical PDF on the filesystem
        '8040701008', '3200040702D', '4311T', ...
      )

    ORDER BY id
    LIMIT 20000 -- The limit is applied AFTER removing duplicates and missing files
)
```

Finally, the third logical block performs the necessary relational joins to gather the exhaustive technical metadata associated with each cleaned article. The retrieval of this specific information is not arbitrary; these database fields represent the **ground truth**, the correct answer key, that the LLMs will be trained to autonomously predict and extract from the chaotic text of the technical drawings. To construct this comprehensive target output, the query systematically extracts a precise set of variables for each document: the unique component code, the general description, the derived component type, the specific raw material, the finished piece weight, the cutting methodology, and a consolidated string of all the required mechanical processes. Most notably, to handle parameters that fluctuate significantly between different mechanical parts, this final `SELECT` statement utilizes JSON aggregation functions (`JSON_AGG` and `JSON_BUILD_OBJECT`) to package variable mechanical characteristics into a structured format, preparing the data for the subsequent JSONL serialization phase.

Code 3.4: Filtering Query - Third logical block: Relational joins and JSON metadata aggregation

```
-- 3. FINAL METADATA RETRIEVAL
SELECT
    ap.codice,
    ap.descrizione,
    ap.nome_file_pdf,
    ap.formato_file,
```

```
    SPLIT_PART(ap.descrizione, ' ', 1) AS tipo_componente,
    ap.versione,
    ap.data_versione,
    m.descrizione AS materiale,
    cf.peso_netto_reale AS peso_pezzo_finito,
    CASE
        WHEN dp.ignora_taglio THEN 'NON PREVISTO'
        WHEN dp.taglio_mat THEN 'TAGLIO MATERIALE'
        ELSE 'A BARRA'
    END AS taglio_materiale,
    STRING_AGG(distinct lav.descrizione, ', ') AS lavorazioni,
    JSON_AGG(
        JSON_BUILD_OBJECT(
            'caratteristica', cfc.descrizione,
            'valore', dc.valore_caratt
        )
    ) FILTER (WHERE dc.valore_caratt IS NOT NULL) AS caratteristiche_json

FROM articoli_puliti ap

LEFT JOIN caratteristiche_fisiche_articolo cf ON cf.id = ap.id
LEFT JOIN tipo_materia_prima tm ON tm.id = cf.id_tipologia
LEFT JOIN materiale m ON m.id = cf.id_materiale
LEFT JOIN distinta_produzione dp ON dp.id = ap.id
LEFT JOIN dettaglio_lavorazioni_dp dl ON dl.id_articolo = ap.id
LEFT JOIN articolo lav ON lav.id = dl.id_lavorazione
LEFT JOIN dettaglio_caratteristiche dc ON dc.id_articolo = cf.id AND dc.nome_caratt <> 'scarCaratt
    '
LEFT JOIN configuratore_caratteristiche cfc ON cfc.id_tipologia = cf.id_tipologia AND LOWER(cfc.
    codice) = LOWER(dc.nome_caratt)

GROUP BY
    ap.id, ap.codice, ap.descrizione, ap.versione, ap.data_versione,
    ap.nome_file_pdf, ap.formato_file,
    m.descrizione, cf.peso_netto_reale, dp.ignora_taglio, dp.taglio_mat
ORDER BY ap.id;
```

3.1.2 Parallel Download and Integrity Verification

After successfully isolating the target documents through the database filtering query, the next step was the data transfer. Downloading twenty thousand PDF files sequentially would have taken a very long time, making the process vulnerable to network interruptions and risking saturating PMP's upload network. To overcome this limitation, the decision was made to implement a **multithreaded**

architecture.

Rather than relying on a sequential approach, the Python script divides the definitive list of documents into ten distinct sub-groups, assigning the task to ten virtual workers to download the files simultaneously. Each thread establishes its own dedicated SSH connection based on the *Paramiko* library to manage the secure tunnel efficiently. Establishing separate connections allows the number of active threads to be dynamically adjusted. This flexibility ensures that data traffic can be managed effectively, by activating or deactivating threads, without saturating the network or slowing down PMP's internal daily operations.

Furthermore, the download engine implements a *fallback* mechanism: if the server cannot find a document at the expected remote path, the script automatically attempts to locate it using a lowercase filename extension before definitively marking it as missing and giving up.

Code 3.5: Multi-threading download mechanism with independent SSH connections and resilient fallback

```
def download_chunk(df_chunk):
    # Dedicated SSH connection for this specific thread
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    my_key = paramiko.Ed25519Key.from_private_key_file(KEY, password=KEY_PASSWORD)
    ssh.connect(HOSTNAME, username=USERNAME, pkey=my_key)
    sftp = ssh.open_sftp()

    for index, row in df_chunk.iterrows():
        nome_file_grezzo = str(row['nome_file_pdf']).strip()
        codice_articolo = str(row['codice']).strip()

        local_filename = f"{codice_articolo}{os.path.splitext(nome_file_grezzo)[1]}"
        local_path = os.path.join(LOCAL_DEST_FOLDER, local_filename)
        remote_path = f"{REMOTE_BASE_PATH}/{codice_articolo}/{nome_file_grezzo}"

        if not os.path.exists(local_path):
            try:
                sftp.stat(remote_path)
                sftp.get(remote_path, local_path)
            except FileNotFoundError:
                # Resilience: Fallback to lowercase search
                remote_path_lower = f"{REMOTE_BASE_PATH}/{codice_articolo}/{nome_file_grezzo.lower
                    ()}"
                try:
```

```
            sftp.stat(remote_path_lower)
            sftp.get(remote_path_lower, local_path)
        except FileNotFoundError:
            # Logging the missing file safely across threads using threading.Lock
            with print_lock:
                with open(LOCAL_NOT_FOUND, "a") as f:
                    f.write(f"'{codice_articolo}', ")
```

However, quickly transferring files was only half of the objective; after downloading, they needed to be checked to ensure their integrity and correctness. Prior to initiating the download phase, the system executes preventive scripts to identify potential naming collisions, instances where two different mechanical components attempt to download a file with the identical local name, causing the system to intercept it to avoid destructive overwrites.

To manage the unpredictability of PMP's file system, the script was designed to generate a series of segregated **control logs** during execution, such as lists of duplicate documents, missing items, and runtime errors. The generation of these distinct files serves a precise purpose: instead of allowing a missing document or a corrupted byte stream to crash the entire multi-threaded pipeline, the system records the anomaly in the respective log and continues its operation. These logs act as feedback loops to trace *silent failures* and refine the dataset.

Following the completion of the download threads, the pipeline executes a final cleanup protocol to purge the local folder of any unauthorized files that were not explicitly requested by our official list. Finally, a structural integrity check is performed: every single downloaded PDF is virtually opened to verify that the file is not corrupted or empty after the download. If the system detects any corrupted, truncated, or empty documents during this inspection, it discards them. To ensure the dataset maintains its required volume, the pipeline cyclically re-executes the database filtering query to identify new valid candidates. It then automatically triggers a new download cycle for these replacement files, repeating this iterative process until the exact quota of healthy documents is reached.

The ultimate output of this rigorous data acquisition pipeline is the creation of the definitive **ground truth** Comma-Separated Values (CSV) file. As discovered during the process, this file differs from the initial export because it is the result of a remediation: it contains exclusively the twenty thousand files that successfully survived every existence and integrity check. This data collection represents the foundational pillar required to transition to the next macro-phase of the

project, where the physical documents will be processed and serialized for the AI models.

3.2 Dataset Generation

Once the physical acquisition of the documents was finalized, the project entered the **Dataset Generation** phase. While the previous stage ensured we had the necessary raw files, the challenge now shifted toward transforming these unstructured data sources into a training corpus. In the context of LLMs, the performance of the fine-tuning process is directly proportional to the cleanliness and structural integrity of the input data. Therefore, this phase was designed as a multi-stage *pipeline* dedicated to distilling raw visual information into a structured, instruction-based format that the models could effectively learn from.

The core objective of this generation process is to bridge the gap between human-readable engineering drawings and machine-readable text. Mechanical PDF files are difficult to parse because they are not always digitally native; many are flat image scans or contain complex vector graphics that standard text readers cannot interpret. To solve this, we developed a hybrid *parser* that evaluates each document to determine the most efficient extraction method. By combining *native text processing* for digital files with OCR for scanned images, the script attempts to correctly read as much data as possible, saving and structuring it in a standardized format.

However, extracting raw text is merely the first step. To enable **Instruction Tuning**, the extracted information must be paired with the corresponding **Ground Truth** retrieved during the database filtering stage. This alignment process is what allows the model to understand the relationship between a chaotic technical drawing and its structured specifications. During this phase, the data undergoes a process of **Prompt Engineering**, where it is wrapped into specific instruction templates designed to simulate a dialogue between a user and an AI assistant. This ensures that the model learns not just to read, but to follow specific formatting constraints and industrial logic.

The final stage of this macro-phase involves the technical serialization of the data. The project adopts the JSONL format, which allows for efficient streaming of thousands of examples during the *tokenization* process. By the end of this generation pipeline, the initial collection of twenty thousand documents is transformed into a definitive set of training, validation, and test datasets.

The following subsections will provide a detailed technical breakdown of these operations.

Section **Text Extraction: Native Processing and Optical Character Recognition** will examine the hybrid extraction script; Section **Ground Truth Construction** will detail the metadata alignment logic; Section **Prompt Engineering and Instruction Format** will discuss the development of instruction formats; and finally, Section **Splitting and JSONL Serialization** will describe the serialization and final dataset partitioning strategies.

3.2.1 Text Extraction: Native Processing and Optical Character Recognition

The integrity of the final dataset depends entirely on the accuracy of the initial **text extraction** process. Mechanical technical drawings retrieved from the PMP file system have a heterogeneous nature; they range from native digital PDF files, where the document is automatically generated by the program that created the digital version of the component, to files containing only low-resolution paper scans digitized into flat images depicting the object. The challenge was therefore to create a Python script that could obtain as much correct information as possible regardless of the content of the document it analyzes, so that it could provide the LLM with as much information as possible during training and inference. The idea was therefore to create a hybrid script, capable of evaluating the internal structure of each document to select the most appropriate parsing methodology, combining native Python libraries for extracting metadata from PDFs with OCR libraries for scanning documents.

The first line of processing is the **Native Text Processing** mode. This technique is reserved for documents that possess a genuine text layer, where characters and coordinates are already embedded in the file's metadata. Fundamentally, a native extraction library operates by directly parsing the underlying structure of the PDF rather than analyzing its visual appearance. It reads the internal data streams and dictionaries where the original software encoded the characters and their precise spatial positioning, allowing the retrieval of text with minimal computational overhead. For this task, the project utilizes the *pypdfium2* library, which has demonstrated excellent performance and accuracy in rapidly extracting information from raw strings across multiple pages. As illustrated in code 3.6, the system attempts to open the document and iterate through its pages to aggregate all available text into a single buffer.

Code 3.6: Implementation of the native text extraction using the pypdfium2 library

```python
import pypdfium2 as pdfium
import os

def extract_native_text(pdf_path):
    text_content = ""
    try:
        pdf = pdfium.PdfDocument(pdf_path)
        for i in range(len(pdf)):
            page = pdf.get_page(i)
            textpage = page.get_textpage()
            text_content += textpage.get_text_range() + "\n"
            textpage.close()
            page.close()
        pdf.close()
    except Exception as e:
        print(f"Errore lettura nativa su {os.path.basename(pdf_path)}: {e}")
        return ""
    return text_content.strip()
```

The second stage of document processing is the **OCR** mode: if the document contains an image of the piece without metadata, then native text processing would not be able to extract anything from the PDF, thus making a library that uses computer vision as a remedy necessary. Fundamentally, an OCR library is a software tool designed to analyze digital images, recognize the shapes of printed letters or numbers, and convert those visual patterns back into machine-readable, editable text strings. This process is significantly more computationally expensive and follows a multi-step pipeline. As shown in code 3.7, the PDF page is rasterized into a bitmap image and significantly scaled up using a magnification matrix, which was set to a zoom factor of 5.55 after several extraction tests. This choice was made to ensure that even the smallest annotations are readable, without requiring excessive calculation time for each individual document.

Subsequently, the *Tesseract* engine is employed to read the pixels. Originally developed at Hewlett-Packard between 1984 and 1994 before becoming a highly accurate open-source OCR engine [16], Tesseract utilizes a multi-stage architecture. Instead of relying on a single pass, its internal pipeline performs computer vision operations such as line finding, baseline fitting, word recognition, and adaptive character classification to robustly interpret complex images [16]. To maximize this potential on mechanical drawings, after several tests, the engine is specifically configured with the parameters `-oem 3 -psm 6`, forcing it to use its LSTM and to treat the visual input as a uniform block of text.

Code 3.7: Implementation of the OCR vision pipeline using PyMuPDF and Tesseract

```
import fitz  # PyMuPDF
from PIL import Image
import pytesseract

def ocr_analyzer(pdf_path: str, lang: str = "ita+eng") -> str:
    document = fitz.open(pdf_path)
    zoom = 5.55                          # Zoom scaling for small texts
    matrix = fitz.Matrix(zoom, zoom)     # Matrix scaling the image relative to zoom

    page = document[0]  # Assuming single-page PDF
    pix = page.get_pixmap(matrix=matrix, alpha=False)   # Rasterize to bitmap
    img = Image.frombytes("RGB", (pix.width, pix.height), pix.samples)

    # OCR with pytesseract on the image
    txt = pytesseract.image_to_string(img, lang=lang, config="--oem␣3␣--psm␣6")

    document.close()
    return clean_text(txt)
```

After running the extraction attempts, the script must decide which text to trust. The true intelligence of the pipeline lies in its **decision logic**, which acts as an autonomous gatekeeper. The script evaluates the length of the string obtained from the native extraction: if it is excessively short, especially if it contains fewer than 50 characters, the document likely contains only a scanned image. In this case, the script considers the native extraction a failure and uses the OCR output exclusively, inserting an empty vector in place of the native extraction rather than inserting completely incorrect data that would only cause noise.

Conversely, if the OCR fails entirely, it defaults to the native text. In the optimal event that both methodologies successfully extract a substantial amount of text, the system concatenates both outputs, providing the subsequent LLM processing stage with the most comprehensive context possible.

Code 3.8: The hybrid decision logic for evaluating and combining native parsing and OCR outputs

```
    # Case A: Poor native text, use OCR
    if len(text_extracted.strip()) < 50 and len(ocr_extracted) > 0:
        full_text = ocr_extracted
        print(" -> Using OCR Output (Native text scarce)")

    # Case B: Only native text available
```

```
elif len(ocr_extracted) == 0:
    full_text = text_extracted

# Case C: Both available, pass both to the LLM
else:
    full_text = f"--- NATIVE TEXT ---\n{text_extracted}\n\n--- OCR TEXT ---\n{ocr_extracted}"
```

This specific implementation is deliberately designed to prioritize system resilience and resource management. By utilizing a context manager, the `with` statement, the script guarantees that the file stream is automatically closed after reading. Furthermore, the explicit inclusion of `try-except` blocks throughout the extraction process represents an architectural safeguard in case of file reading or scanning errors. Once the final text is selected, it passes through cleaning functions to remove encoding artifacts, preparing a normalized string ready to be paired with the structured ground truth.

3.2.2 Ground Truth Construction

One of the most important aspects of making an AI model operate correctly is the availability of high-quality training labels. In this project, the **Ground Truth** represents the ideal answer we want the model to generate when it processes a technical drawing. While the previous extraction phase provided the raw text to work with, creating these reference labels is what actually teaches the system how it is supposed to behave. This process is fundamental for SFT, whose task is to refine the model to meet the user's specific goals [23]. Building upon traditional SFT, which often relies on simple (instruction, output) pairs, this dataset utilizes modern conversational *triplets* based on **Instruction Formatting**, composed of system, user, and assistant messages. By providing these explicit instructional examples, we ensure that the model does not just predict the next word in a generic sequence, but actively adheres to the formatting standards of the task for which it is trained, in this case adhering to the format of the mechanical documentation produced by PMP.

To build this reference dataset, the system utilizes the metadata successfully validated during the initial database query. Each record in our finalized CSV file contains a precise set of attributes: the component code, the technical description, the material, the required mechanical processes, and the specific cutting methodologies. The challenge was to transform these isolated database fields into a coherent, natural language *target sequence* that mimics a professional report. This is

an important step because providing a clear and consistent structure helps the model to generalize better and reduces the risk of *label noise*, which occurs when ambiguous or inconsistent training labels degrade the final performance [7].

The implementation of this logic is handled within the main dataset generation script, specifically through the construction of the **assistant response**. As shown in code 3.9, the script maps the database variables into a standardized textual template. This template is designed to provide an information summary, explicitly stating the relationship between the component and its specifications. This structured approach is inspired by the *JsonTuning* methodology, which suggests that explicitly defining the relations between task elements leads to more robust and controllable model outputs [7].

Code 3.9: Mapping of validated database metadata into the standardized assistant response template

```
# Assistant output construction: the ground truth
assistant_content = (
    f"Analisi del componente {codice} ({descrizione}): "
    f"Il documento contiene specifiche tecniche compatibili con la descrizione. "
    f"MATERIALE: {materiale}. "
    f"LAVORAZIONI: {lavorazioni}. "
    f"TIPO DI COMPONENTE: {tipologia}. "
    f"TAGLIO MATERIALE: {taglio}"
)
```

From an architectural standpoint, constructing the ground truth in this manner serves two strategic purposes. First, it enforces a **standardized output** format that the model will learn to replicate during inference, ensuring that the extracted data is always presented in the same order and with the same terminology. Second, it allows for a mathematically rigorous evaluation during the testing phase. Because every training example has a corresponding, verified ground truth derived directly from the corporate Enterprise Resource Planning (ERP), we can objectively measure the distance between the model's predictions and the actual industrial facts. This alignment between the visual document and the verified metadata constitutes the foundational training signal that enables the LLM to function as a reliable autonomous parser.

3.2.3 Prompt Engineering and Instruction Format

In order to go from the raw text extracted from documents to a dataset ready for training, it was necessary to apply a **Prompt Engineering** process, that is, the design, optimization, and logical structuring of the prompts provided to an AI model in order to guide its behavior, maximize its accuracy, and obtain relevant and structured outputs. This phase is the core of **Instruction Tuning**, a methodology that specializes LLMs by demonstrating exactly how to respond to specific user requests [23]. For this project, the goal was to create a robust and repeatable *instruction template* that forces the model to act as a "specialized mechanical engineer", as indicated in the prompt provided to the model, ensuring that the extracted data is not only accurate but also formatted according to the requirements of Studio ITC and PMP.

The architecture of each training example follows a conversational structure based on **Instruction Formatting**. Unlike traditional fine-tuning that might only use simple input-output pairs, as in the standard SFT, this project utilizes a **triplet** configuration composed of three distinct roles: the **System Prompt**, the **User Prompt** and the **Assistant Prompt**.

The **System Prompt** serves as the primary behavioral constraint, defining the identity and the operational boundaries of the model.

Code 3.10: Definition of the System Prompt that defines the professional personality of the model

```
# System Prompt: Defining the behavioral persona
system_prompt = (
    """Sei un assistente esperto il cui scopo è estrarre le specifiche tecniche del componenete
    meccanico. Per farlo devi analizzare il contenuto estratto dall'analisi nativa del testo
    nativo e dalla scansione OCR."""
)
```

By explicitly telling the model that it is an expert assistant specialized in reading mechanical drawings, we prime its internal weights to prioritize technical terminology and industrial logic over generic linguistic patterns. This specialized "persona" is what allows the model to ignore irrelevant PDF noise and focus exclusively on the technical extraction task. Furthermore, as defined in the code string, the system is explicitly instructed to base its analysis on both the native text extraction and the OCR scan, establishing a solid foundation that enables the model to utilize a dual-reading strategy.

Following the system initialization, the **User Prompt** provides the actual task-specific data and

the explicit list of extraction goals.

Code 3.11: Structuring the User Prompt with specific technical requirements and extracted text

```
# User Prompt: Structuring the task and providing the raw data
user_prompt = f"""
Analizza il seguente documento tecnico associato al componente meccanico.

METADATI DA ESTRARRE:
- Codice componente
- Descrizione
- Tipo di componente
- Materiale dichiarato
- Taglio materiale
- Lavorazioni

CONTENUTO ESTRATTO DAL DOCUMENTO (Include testo nativo e scansione OCR):
{extracted_content}

RICHIESTE:
1) Analizza il testo estratto nativamente ed estratto tramite OCR.
2) Identifica e compila i metadati mancanti basandoti SOLO sul contenuto del documento.
3) Sintetizza le caratteristiche tecniche principali.
"""
```

A structural key in this phase was to provide the model with the combined `{extracted_content}` variable, representing both the native text and the OCR output simultaneously. This information stream allows the AI to cross-reference data points, effectively using the clarity of the native layer to correct potential *character recognition errors* from the OCR scan. Furthermore, the inclusion of a specific list of "*METADATI DA ESTRARRE*" with explicit requests acts as a structural anchor. This approach is strongly supported by research into *JsonTuning*, which suggests that providing a clear, structured list of expectations reduces model ambiguity and improves the controllability of the output [7]. Forcing the model to rely "*SOLO sul contenuto del documento*" helps to counteract possible hallucinations of the model, ensuring the AI does not invent any parameters.

The final component of the conversational triplet is the **Assistant Prompt**, which embodies the explicit *ground truth* for the extraction task. While the user prompt provides the problem, the assistant prompt provides the perfect solution.

Code 3.12: Definition of the Assistant Prompt, representing the structured ground truth

```
# Assistant Prompt: The structured ground truth output
assistant_prompt = (
    f"""Analisi del componente {codice} ({descrizione}): il documento contiene specifiche tecniche
         compatibili con la descrizione.
    MATERIALE: {materiale}. LAVORAZIONI: {lavorazioni}. TIPO DI COMPONENTE: {tipologia}. TAGLIO
        MATERIALE: {taglio}"""
)
```

This specific string formulation dictates the target behavior of the LLM. By injecting the validated database variables, such as `{materiale}` and `{lavorazioni}`) into a fixed textual template, we construct a standardized response. This teaches the model not only *what* data to extract from the chaotic input, but exactly *how* to format it. Consistently presenting the output in this rigid structure across all documents ensures uniformity, heavily reducing hallucinations and making the model's responses predictable and easily parsable by downstream software.

Once all three roles are defined, they must be assembled into a standard architecture that the training algorithms can process. This is achieved through JSON Serialization, which explicitly maps each text block to its specific conversational role.

Code 3.13: Packaging the conversational triplet into the final JSON record format

```
# Packaging the triplet into the final JSON record format
json_record = {
    "messages": [
        {"role": "system", "prompt": system_prompt},
        {"role": "user", "prompt": user_prompt},
        {"role": "assistant", "prompt": assistant_prompt}
    ]
}
```

By packaging these three prompt elements into a single JSON dictionary, we create a complete unit for the model, from which it can learn and be trained. The model learns that whenever it assumes the expert system persona and receives a chaotic technical drawing in the user prompt, it is required to generate the structured mechanical report found in the assistant field.

3.2.4 Splitting and JSONL Serialization

The final operation in the dataset preparation pipeline is the mathematical partitioning of the data and its serialization. A model's true utility is measured not by its performance on familiar data,

but by its *generalization capability* on unseen instructions and tasks [23]. To ensure this, it is mandatory to keep the information used for learning separated from the data used for evaluation. To prevent **overfitting**, a condition where the model merely memorizes the training examples without developing the underlying logic to process new technical drawings, the project implements a **Dataset Splitting** strategy. The documents are thus divided into three mutually exclusive subsets: the **Training Dataset**, the **Validation Dataset** and the **Test Dataset**.

The distribution of data in the three datasets follows a 70/15/15 ratio. The **Training Dataset**, 70% of the data, is the primary resource for the weight adjustment process during fine-tuning. The **Validation Dataset**, 15% of the data, acts as a secondary control loop, allowing for the real-time monitoring of loss functions and the optimization of hyperparameters. Finally, the **Test Dataset**, 15% of the data, is preserved as a blind benchmark, entirely unseen by the model during the training phase, to ensure an objective final evaluation of its generalization skills.

To maintain statistical validity, the script performs a stochastic **shuffling** of the dataframe before partitioning, ensuring that the distribution of component types and material specifications remains uniform across all subsets.

Code 3.14: Implementation of the datasets splitting logic

```
# Shuffling the data to ensure statistical independence
df_shuffled = df.sample(frac=1, random_state=7).reset_index(drop=True)

# Calculating split indices based on predefined ratios
total_rows = len(df)
train_end = int(total_rows * RATIO_TRAINING_SET)
val_end = train_end + int(total_rows * RATIO_VALIDATION_SET)

# Partitioning the dataframe
df_train = df_shuffled.iloc[:train_end]
df_val = df_shuffled.iloc[train_end:val_end]
df_test = df_shuffled.iloc[val_end:]
```

From a data serialization perspective, the project adopts the JSONL format, since representing the training tasks through explicit JSON structures guarantees high fidelity and reduces parsing ambiguities during model training [7]. However, while a standard JSON array would require loading the entire dataset into the system's memory simultaneously, the JSONL architecture treats each conversational triplet as a standalone, independent text line.

By serializing the records line-by-line, the training algorithm can stream the data iteratively. This approach effectively manages the Video Random Access Memory (VRAM) of the GPUs, preventing memory overflow bottlenecks that would inevitably occur if the system attempted to process thousands of reports all at once. To ensure structural integrity during writing, each record is serialized utilizing a custom encoding utility, specifically a `NumpyEncoder` class, designed to safely handle complex numerical data types and ensure proper 8-bit Unicode Transformation Format (UTF-8) character preservation.

This meticulous process generates the definitive partitioned files: `dataset_training.jsonl`, `dataset_validation.jsonl` and `dataset_test.jsonl`. The creation of these pristine files marks the completion of the data acquisition and generation macro-phase.

3.3 Fine-Tuning

After completing the data generation pipeline, the next step is the most computationally intensive phase of the project: the execution of the **Fine-Tuning** process. While the previous chapters focused on the acquisition and structural normalization of industrial metadata, this section details the architectural strategy employed to transform generic LLMs into specialized extractors for mechanical documentation. The primary objective is to perform an SFT that enables the models to internalize the relationships between raw technical text, OCR artifacts, and the structured ground truth derived from the ERP system of PMP.

To achieve this goal, the project adopts the PEFT paradigm. In contrast to full fine-tuning, which requires updating all parameters of a model and demands prohibitive computational resources, PEFT focuses on modifying only a small subset of additional weights. This approach has been adopted on two macro families of LLMs, ranging from 7 to 14 billion parameters. Specifically, the research focuses on the implementation and comparison of two techniques: LoRA, which injects trainable rank decomposition matrices into the Transformer layers [9], and its quantized evolution, QLoRA, which enables training on 4-bit compressed weights [4].

The experimental framework is designed as a comprehensive comparative study involving twelve distinct training sessions. The methodology consists of testing six different model families, including the Llama, Mistral, Qwen, and Phi architectures, under both LoRA and QLoRA modalities. This

comparison is intended to evaluate not only the final extraction accuracy but also the computational efficiency, training time, and memory footprint of each configuration. This setup allows for an objective assessment of the *scaling laws* and the trade-offs between quantization and performance in a real-world industrial context.

The following sub-chapters will provide a detailed breakdown of the technical implementation. First, Section **High Performance Computing Environment and Model Selection** describes the HPC environment and the criteria used for model selection. Section **Memory Optimization** explains the memory management strategies necessary to handle large models on the available hardware. Section **LoRA and QLoRA Implementations** details the specific configurations of the adapters, while Section **Training Hyperparameters** summarizes the definitive set of training hyperparameters derived from the analysis of the training scripts.

3.3.1 High Performance Computing Environment and Model Selection

The computational feasibility of fine-tuning large-scale models is heavily dependent on the available hardware infrastructure and the specific selection of base architectures. The initial activities of the project were carried out using **Runpod**, a cloud-based GPU rental platform. This setup facilitated the rapid testing of the prototype extraction pipeline and the refinement of training scripts. It provided a flexible, on-demand workspace to conduct preliminary tests and verify technical feasibility before scaling up the experiments.

Once the methodology was stabilized, the entire production workload was migrated to the HPC cluster of **UniMoRe**, the University of Modena and Reggio Emilia. The university cluster offered a more robust and scalable infrastructure with a wide array of hardware options. For the fine-tuning phase, a single NVIDIA A40 GPU equipped with 48 GB of VRAM was selected. This hardware choice acted as an architectural constraint, determining the maximum number of model parameters that could be processed in a single training session, thus striking a balance between high performance and resource efficiency for implementation in small or medium-sized enterprises.

The selection of the LLMs followed a market-driven criterion, prioritizing the most widely adopted and prominent open-weights models available on the *HuggingFace* platform. To ensure a comprehensive and scientifically valid comparison, the project focused on selecting one representative model from each major high-performing family, namely **Llama** from Meta, **Mistral**

from Mistral AI, **Phi** from Microsoft, and **Qwen** from Alibaba. The research design is structured to facilitate an equitable comparison across two different scales of magnitude. Specifically, four models were selected from the **7B** and **8B** parameter class, including *Llama-3.1-8B*, *Mistral-7B-v0.2*, *Phi-3-Small*, and *Qwen2.5-7B*, while two models were selected from the **14B** parameter class, specifically *Phi-4* and *Qwen2.5-14B*. This distribution was deliberately chosen to investigate how different architectural approaches influence the extraction accuracy within the same parameter range [18], while keeping the overall size of the models manageable for a single-GPU setup.

The decision to cap the model size at 14 billion parameters was a deliberate choice aimed at maintaining system resilience on the available hardware. By focusing on the 7B-14B range, the project could leverage the full potential of **LoRA** and **QLoRA** adapters on a single NVIDIA A40, allowing for a deep dive into the trade-offs between model scale and quantization efficiency. This setup ensures that the results are not only accurate for the specific industrial use case of PMP but also reproducible and representative of the current computational efficiency trends in the NLP landscape [2].

3.3.2 Memory Optimization

The primary challenge in fine-tuning LLMs, especially with hardware constraints such as a single NVIDIA A40, is the management of the VRAM footprint. For models in the 7B to 14B parameter range, the combined memory requirements of the model weights, optimizer states, and gradients can easily exceed the physical capacity of the GPU. To ensure architectural stability and prevent **OOM** errors, the project implements a multi-layered **Memory Optimization** strategy based on techniques in the NLP field.

One of the fundamental pillars of this optimization is the use of **Gradient Accumulation**. As configured in the training pipeline, instead of updating the model weights after every individual batch, the system calculates gradients over multiple smaller micro-batches and aggregates them before performing a single optimization step. This approach allows the project to achieve a large **effective batch size** while maintaining a low instantaneous memory overhead. As demonstrated in the QLoRA framework, decoupling the physical memory constraint from the mathematical batch size is a strategy to maintain stable convergence when fine-tuning large models on contained hardware [4]. Specifically, the project set the physical batch size to **4** for the **7B** and **8B** parameter

models with a gradient accumulation of 8, resulting in an effective batch size of **32**. For the **14B** parameter models, however, the batch size was strictly set to **1** with a gradient accumulation of **32** to accommodate and prevent OOM errors. This guarantees the exact same effective batch size of **32** across all experiments, ensuring that the gradient descent updates remain mathematically equivalent and statistically stable without saturating the VRAM. This adaptive configuration is implemented in the training script as shown in code 3.15.

Code 3.15: Dynamic allocation of batch size and gradient accumulation based on model size

```
# Custom parameters based on model family
model_name_lower = args.model_name.lower()
is_14b = "14b" in model_name_lower or "phi-4" in model_name_lower

if is_14b:
    print(f"--- MODEL 14B ---")
    args.per_device_train_batch_size = 1
    args.gradient_accumulation_steps = 32
    args.learning_rate = 0.0001
else:
    print(f"--- MODEL 8B ---")
    args.per_device_train_batch_size = 4
    args.gradient_accumulation_steps = 8
    args.learning_rate = 0.0002
```

Furthermore, the implementation utilizes **Paged Optimizers**, specifically the `paged_adamw_8bit` variant. Paged optimizers leverage the NVIDIA unified memory feature to handle memory spikes by offloading optimizer states to the Central Processing Unit (CPU) Random Access Memory (RAM) when the GPU memory is near capacity [4]. This is combined with the adoption of BF16 precision, which provides a higher dynamic range than standard float16, effectively reducing memory usage by about 50% compared to full 32-bit precision while maintaining the numerical stability required for deep learning on technical documentation [4].

During the training phase, a specific security measure was adopted for the *Flash Attention* mechanism, particularly for the Phi-3 and Phi-4 architectures. As shown in code 3.16, specific patches were applied to handle type casting between float32 and bfloat16, preventing crashes during the processing of long sequences within the attention layers.

Code 3.16: Patches for Flash Attention and memory casting in the training script

```
# FIX for flash attention: Type casting to avoid QLoRA crashes
```

```
try:
    import flash_attn.flash_attn_interface
    _original_varlen_func = flash_attn.flash_attn_interface.flash_attn_varlen_func

    def safe_flash_attn_varlen_func(q, k, v, *args, **kwargs):
        # Convert float32 input sequence to bfloat16 for stability
        if q.dtype == torch.float32: q = q.to(torch.bfloat16)
        if k.dtype == torch.float32: k = k.to(torch.bfloat16)
        if v.dtype == torch.float32: v = v.to(torch.bfloat16)
        return _original_varlen_func(q, k, v, *args, **kwargs)

    flash_attn.flash_attn_interface.flash_attn_varlen_func = safe_flash_attn_varlen_func
except ImportError:
    pass
```

Finally, the project controls the **Max Sequence Length**, setting a limit of 2048 tokens. This constraint is vital for resource management, as the memory complexity of the attention mechanism increases quadratically with the sequence length. By striking a balance between context window size and hardware limitations, the project ensures that any document containing mechanical drawings can be processed efficiently, allowing for the execution of all twelve training experiments within the HPC environment's resource quotas.

3.3.3 LoRA and QLoRA Implementations

The core of the project's PEFT strategy lies in the implementation of **LoRA** and its quantized variant, **QLoRA**. As discussed in **LoRA: Low-Rank Adaptation** and **QLoRA: Quantized Low-Rank Adaptation**, the primary goal of these techniques is to adapt LLMs to specific tasks without the necessity of retraining the entire weight matrix. Mathematically, for a pre-trained weight matrix $W_0 \in \mathbb{R}^{d \times k}$, LoRA represents the update by a low-rank decomposition $W_0 + \Delta W = W_0 + BA$, where $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$ are trainable matrices of rank $r \ll \min(d, k)$ [9].

In this project, the configuration of the LoRA adapters was engineered to make data extraction from documents as accurate as possible while adhering to hardware limitations. To achieve this balance between model capacity and computational efficiency, two hyperparameters must be defined: the rank r and the scaling factor α. The rank r dictates the dimensionality of the low-rank matrices, effectively controlling how many new parameters are injected into the model for training [9]. The parameter α acts as a constant multiplier; it scales the magnitude of the learned low-rank updates

(ΔW) before they are added to the pre-trained weights, essentially determining how much influence the new task-specific learning has compared to the original model knowledge [9].

As shown in code 3.17, the core parameters driving the adaptation in the `train.py` script are a **rank** $r = 16$ and a **scaling factor** $\alpha = 32$.

Code 3.17: Configuration of the LoRA adapters in the training script

```
# Defining LoRA hyperparameters
lora_rank = 16
lora_alpha = 32

lora_config = LoraConfig(
    r=lora_rank,
    lora_alpha=lora_alpha,
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj", "down_proj"],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM"
)
model = get_peft_model(model, lora_config)
```

A **rank** value of 16 was selected to provide the model with a higher degree of representational freedom compared to the minimum requirements typically suggested by literature. Empirical studies on LoRA indicate that low-rank adaptation is remarkably effective even with very small ranks, such as $r = 4$ or $r = 8$, which are often sufficient to capture the essential task-specific information without significant loss in performance [9]. However, for this project, a rank of 16 was implemented to ensure the model possessed the necessary capacity to internalize the complex formatting, the strict JSON structural rules, and the specific technical jargon of the PMP documents, while still maintaining a manageable VRAM footprint. To complement this, setting $\alpha = 32$ establishes a standard $2:1$ ratio with the rank. This configuration acts as a controlled amplifier for the adapter's gradients, ensuring a stable learning process and allowing the model to adapt to the rigid structure of the ground truth without experiencing catastrophic forgetting of its foundational linguistic capabilities [9].

Furthermore, unlike early implementations that only targeted the *Attention* layers (Q, V), this project extends the adaptation to **all linear modules** within the Transformer architecture. According to the QLoRA methodology, applying adapters to all linear transformer block layers is fundamental for achieving performance parity with full fine-tuning [4]. The specific target modules defined in the configuration represent the fundamental mathematical operations of the model:

- **`q_proj`, `k_proj`, `v_proj`:** These matrices project the input into *Queries*, *Keys*, and *Values*, driving the Self-Attention mechanism that allows the model to understand the contextual relationships between words in the technical documents.
- **`o_proj`:** The *output projection matrix* that aggregates the results of the attention mechanism.
- **`gate_proj`, `up_proj`, `down_proj`:** These matrices constitute the *MLP layers*, responsible for processing the representations generated by the attention mechanism into higher-level features.

Targeting all these layers allows the model to better adapt its internal knowledge representation to the specialized vocabulary for the specific task for which it is trained [4].

When the training mode is set to **QLoRA**, an additional layer of optimization is applied through 4-bit quantization. As shown in code 3.18, the project utilizes the NF4 data type.

Code 3.18: Implementation of 4-bit quantization for QLoRA

```
# QLoRA specific quantization configuration
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,  # Saves memory by quantizing the constants
    bnb_4bit_quant_type="nf4",      # Information theoretically optimal for weights
    bnb_4bit_compute_dtype=torch.bfloat16
)
```

The use of NF4 is strategically significant as it is optimal for normally distributed weights, which is standard for pre-trained LLMs [4]. This is paired with **Double Quantization**, a technique that quantizes the quantization constants themselves, saving about 0.37 bits per parameter [4]. By combining these configurations, the project ensures that the 14B models can be trained with the same level of precision and performance as their 7B counterparts, maintaining the same level of accuracy across all twelve experimental configurations.

3.3.4 Training Hyperparameters

Optimizing the instruction tuning process relies heavily on the selection of **Training Hyperparameters**. Beyond the architectural choices of the adapters and memory management, these parameters govern the dynamics of the gradient descent and the stability of the convergence. For this project,

the hyperparameter configuration was derived from the recommendations provided by the QLoRA and LoRA frameworks, adapted to the specific needs of document processing.

The core of the training logic is encapsulated in the `TrainingArguments` block of the `train.py` script. As shown in code 3.19, the project employs a set of parameters designed to ensure a robust and repeatable learning process across all twelve experimental configurations.

Code 3.19: Configuration of the training hyperparameters

```
training_args = TrainingArguments(
    output_dir=args.output_dir,
    per_device_train_batch_size=args.per_device_train_batch_size,
    gradient_accumulation_steps=args.gradient_accumulation_steps,
    learning_rate=args.learning_rate,
    num_train_epochs=3,
    per_device_eval_batch_size=1,
    eval_strategy="steps",
    eval_steps=50,
    logging_steps=5,
    fp16=False,
    bf16=True,
    optim="paged_adamw_8bit",
    gradient_checkpointing=True,
    save_strategy="steps",
    save_steps=100,
    save_total_limit=2,
    max_grad_norm=0.3,
    warmup_ratio=0.03,
    report_to="none"
)
```

One of the most important parameters is the **Learning Rate**. The project implements a differentiated learning rate strategy based on the number of model parameters, following the empirical approach in the QLoRA framework. As explicitly defined in the training script in code 3.20, for the 7B and 8B models, a learning rate of 2×10^{-4}, equivalent to 0.0002, is used. Conversely, for the more complex 14B models, the rate is systematically reduced to 1×10^{-4}, equivalent to 0.0001 [4].

Code 3.20: Dynamic selection of the learning rate and batch parameters based on model scale

```
# Custom parameters based on model family
model_name_lower = args.model_name.lower()
is_14b = "14b" in model_name_lower or "phi-4" in model_name_lower

if is_14b:
```

```
        print(f"--- MODEL 14B ---")
        args.per_device_train_batch_size = 1
        args.gradient_accumulation_steps = 32
        args.learning_rate = 0.0001
    else:
        print(f"--- MODEL 8B ---")
        args.per_device_train_batch_size = 4
        args.gradient_accumulation_steps = 8
        args.learning_rate = 0.0002
```

This reduction is necessary because the optimal learning rate scales inversely with model size, maintaining numerical stability in deeper architectures during the initial phases of the adaptation and preventing the *exploding gradient*, a phenomenon that occurs during training when back-propagated error gradients accumulate and grow exponentially as they propagate backward from the output layer to previous layers [4].

To further stabilize the training, the project adopts a **Max Gradient Norm** of 0.3 and a **Warmup Ratio** of 0.03 [4]. The gradient norm acts as a *gradient clipping* mechanism, preventing outliers in the gradient calculations from destabilizing the 4-bit quantized weights. Concurrently, the warmup ratio ensures that for the first 3% of the training steps, the learning rate is gradually increased from zero to its target value. This soft start allows the LoRA adapters to initialize their weights without disrupting the foundational knowledge of the pre-trained base model [4].

The training duration was set to **3 Epochs** for all models, aligning with the standard baseline established by the LoRA and QLoRA experiments [4]. This allows the model to have sufficient exposure to the data to internalize the formatting requirements, without inducing a catastrophic forgetting of general language skills or leading to severe overfitting [23].

Finally, the use of the **`paged_adamw_8bit`** optimizer is a innovation of the QLoRA methodology [4]. Beyond the memory savings discussed in Section Memory Optimization, this optimizer is specifically designed to manage memory spikes by offloading optimizer states to the CPU RAM. This ensures that the parameters are managed with the precision required for 4-bit quantization, maintaining the information-theoretical optimality of the *NF4* weights throughout the training.

To provide a clear and comprehensive overview of the entire experimental setup, all the discussed parameters have been aggregated and tracked. Table 3.1 summarizes the exact hyperparameter configurations applied across the twelve distinct training sessions, explicitly highlighting the adaptive strategy used to balance the hardware constraints between the 8B and 14B model families.

Table 3.1: Summary of the training hyperparameters applied across all twelve experimental configurations.

Model	Size	Mode	Learning Rate	Batch Size	Epoch	Rank	Alpha
Mistral	7B	lora	0.0002	4	3	16	32
Mistral	7B	qlora	0.0002	4	3	16	32
Phi3	7B	lora	0.0002	4	3	16	32
Phi3	7B	qlora	0.0002	4	3	16	32
Llama3.1	8B	lora	0.0002	4	3	16	32
Llama3.1	8B	qlora	0.0002	4	3	16	32
Qwen3	8B	lora	0.0002	4	3	16	32
Qwen3	8B	qlora	0.0002	4	3	16	32
Phi4	14B	lora	0.0001	1	3	16	32
Phi4	14B	qlora	0.0001	1	3	16	32
Qwen2.5	14B	lora	0.0001	1	3	16	32
Qwen2.5	14B	qlora	0.0001	1	3	16	32

In conclusion, the orchestration of memory optimization techniques, precise LoRA and QLoRA adapter configurations, and selected hyperparameters successfully established a stable and controlled training environment. By dynamically adapting the physical batch size, gradient accumulation steps, and learning rates to the specific architectural scale of each model, the project ensured mathematical equivalence and hardware resilience across all twelve experiments. This extensive fine-tuning pipeline transformed the base foundation models into technical extractors, successfully concluding the training phase. The resulting adapted weights are now structurally prepared for the final phase of the project: the execution of automated inference and the evaluation of their extraction accuracy on unseen documentation.

3.4 Inference and Evaluation Setup

Following the execution of the twelve supervised fine-tuning experiments, the project transitions from the training environment to the inference and evaluation phase. The ultimate goal of instruction tuning is not merely to minimize the loss function on familiar training data, but to ensure that the LLMs can reliably generalize their newly acquired extraction capabilities to unseen instructions and tasks [23]. Therefore, this phase is dedicated to implementing the trained models on the test dataset, testing them in processing data extracted from completely new machine documents and objectively evaluating their outputs with **NLP metrics**.

Operating within a PEFT paradigm introduces a fundamental architectural shift during the inference stage compared to traditional full fine-tuning. As established in the LoRA and QLoRA methodologies, the training process does not alter the original foundation model; instead, it generates a set of lightweight and task-specific adapter weights [9, 4]. Consequently, the inference pipeline requires a modular loading strategy, where the frozen pre-trained base model is loaded into the GPU memory and the corresponding adapters are dynamically injected into its transformer layers.

Furthermore, this project requires the models to perform a constrained task: extracting mechanical metadata and structuring it into programmatically parsable formats. Ensuring that LLMs output valid, controllable, and robust JSON structures requires precise generation strategies and rigorous post-processing mechanisms to mitigate the inherent unpredictability of autoregressive text generation [7]. Once the structured text is safely extracted and parsed, it must be quantitatively compared against the human-annotated ground truth. To achieve this, the pipeline relies on an automated framework utilizing NLP metrics, which evaluate both the exact lexical matches and the semantic similarities of the extracted technical terminology.

The following sub-chapters detail the construction and execution of this automated testing framework. Section **Base Model and Adapter Loading** outlines the technical procedure for loading the base architectures alongside their respective adapters, ensuring memory efficiency. Section **Generation Strategy and Post-Processing** describes the generation hyperparameters and the parsing algorithms utilized to isolate the JSON output from the model's raw text response. Finally, Section **Natural Language Processing Evaluation Metrics Framework** provides a comprehensive overview of the evaluation metrics **BLEU**, **BLEU with smoothing**, **METEOR**, **ROUGE** and

ROUGE-L, employed to mathematically score the extraction accuracy and establish the foundation for the final comparative results of the thesis.

3.4.1 Base Model and Adapter Loading

Unlike traditional fine-tuning where the model weights are permanently altered, the use of PEFT techniques, like LoRA and QLoRA, requires a dual-stage loading logic. This process involves instantiating the frozen foundation model and subsequently injecting the task-specific learned parameters, stored as adapters, into the network's architecture. This modularity is a direct implementation of the LoRA principle, which maintains the pre-trained weights W_0 in a static state while applying the learned updates through separate rank decomposition matrices [9].

The first step in the inference script involves the configuration of the hardware-aware loading strategy. To ensure that the models, particularly the 14B variants, fit within the VRAM of the NVIDIA A40, the system must handle the base architectures with the same precision settings used during training. For QLoRA configurations, this implies loading the foundation model in a 4-bit quantized state. As demonstrated in the QLoRA framework, utilizing the NF4 data type is information-theoretically optimal for the normally distributed weights of pre-trained LLMs, allowing for high-fidelity inference with significantly reduced memory overhead [4].

The technical implementation in the `inference.py` script utilizes the *Hugging Face* `transformers` and `peft` libraries to execute this logic. As shown in code 3.21, the script dynamically defines the `BitsAndBytesConfig` before instantiating the base model with `AutoModelForCausalLM`.

Code 3.21: Implementation of the loading logic for base models and adapters

```
# Configuration for quantized loading in QLoRA mode
bnb_config = None
if mode == 'qlora':
    bnb_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_use_double_quant=True,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_compute_dtype=torch.bfloat16
    )

# Loading the static foundation model
```

```
base_model = AutoModelForCausalLM.from_pretrained(
    hf_model,
    quantization_config=bnb_config,
    device_map="auto",
    trust_remote_code=True,
    attn_implementation="flash_attention_2",
    torch_dtype=torch.bfloat16
)

# Injecting the task-specific LoRA/QLoRA adapters
model = PeftModel.from_pretrained(base_model, adapter_path)
model.eval()
```

Once the base model is loaded, the script performs the step of attaching the **Adapters**. By utilizing the `PeftModel.from_pretrained` method, the system retrieves the low-rank matrices A and B from the checkpoint directory and maps them to the corresponding transformer layers, the *target modules*, identified during the training phase. This operation effectively fuses the general-purpose knowledge of the base model with the specialized mechanical extraction capabilities learned by the adapters [9].

Finally, the model is set to `eval()` mode. This step is mandatory to disable training-specific behaviors, such as dropout, ensuring that the output for a given mechanical document is deterministic and stable. This structured loading sequence ensures that the inference engine operates with maximum efficiency, leveraging the lightweight nature of PEFT to allow for rapid switching between different model families and quantization levels during the evaluation macro-phase.

3.4.2 Generation Strategy and Post-Processing

Once the model and its respective adapters are correctly loaded into memory, the system must execute the **text generation** process. In the context of metadata extraction, the generation strategy is as important as the training itself.

However, LLMs operate as *autoregressive* and *probabilistic* engines, generating text *token-by-token* rather than enforcing hard-coded programmatic rules. This flexibility frequently leads to minor structural hallucinations, such as missing brackets or unescaped characters, and triggers the inclusion of conversational chatter, like introductory preambles or polite concluding remarks. Since automated extraction pipelines require schema compliance, even a single extraneous word

or formatting deviation will instantly crash standard parsing libraries, rendering the raw output computationally unusable without an intermediate correction layer [7]. The goal is to obtain a structured output that strictly adheres to the JSON schema defined during the dataset generation phase.

To mitigate these risks and ensure high-fidelity extraction, the project implements a controlled **Sampling Strategy**. As shown in code 3.22, the `inference.py` script restricts the model's creativity to ensure that it remains firmly anchored to the factual data present in the OCR text.

Code 3.22: Configuration of the generation hyperparameters in the inference script

```
# Execution of the generation process
outputs = model.generate(
    input_ids=inputs.input_ids,
    attention_mask=inputs.attention_mask,
    max_new_tokens=1024,
    do_sample=True,
    temperature=0.1,
    top_p=0.9,
    pad_token_id=tokenizer.eos_token_id
)
```

The parameters defined within the `generate` function govern the autoregressive decoding process, balancing the need for deterministic formatting against the flexibility required to process noisy text:

- **`max_new_tokens = 1024`:** Defines the absolute ceiling for the length of the generated output. This threshold ensures the model possesses an adequate computational workspace to fully articulate the JSON structures, without triggering premature truncation [7].

- **`do_sample = True`:** Activates stochastic decoding. Rather than always selecting the absolute most probable next word, the model samples from a distribution of likely tokens. This is essential when combined with *top-p* to handle the unpredictable linguistic noise typical of OCR artifacts [23].

- **`temperature = 0.1`:** By setting this to an extremely low value, near zero, the model is forced into a highly deterministic state, scaling the logits before the softmax and effectively refining the probability distribution [23]. Low temperature is important for data extraction

tasks where precision and factual adherence to the input text are paramount, suppressing the likelihood of hallucinating ungrounded mechanical properties.

- **`top_p = 0.9`:** Implements *nucleus sampling*, restricting the model to choose only from the smallest set of tokens whose cumulative probability exceeds 90%. This configuration maintains structural reliability while retaining a marginal degree of flexibility, allowing the model to adapt to slightly corrupted input sequences or variations in technical nomenclature [23].

- **`pad_token_id`:** Synchronizes the padding token with the End Of Sequence (EOS) token, a standard requirement in *Hugging Face* pipelines to ensure that sequence generation terminates correctly once the JSON object is closed.

Despite these optimizations, the raw output from the model often requires a **Post-Processing** stage to isolate the actual data. *Structure-to-structure* tasks often suffer from formatting artifacts, such as the inclusion of Markdown code blocks or explanatory text [7]. To handle this, the `inference_function.py` utilizes a robust cleaning pipeline based on **regex** to identify and extract only the content within the outermost curly braces.

Code 3.23: Post-processing logic for JSON extraction and cleaning

```
def clean_output(output_text):
    """
    Cleans the model output to extract only the valid JSON block.
    """
    # Removing potential Markdown code block tags
    output_text = output_text.replace("```json", "").replace("```", "").strip()

    # Utilizing Regex to find the JSON structure
    match = re.search(r"\{.*\}", output_text, re.DOTALL)
    if match:
        return match.group(0)
    return output_text
```

The final step of the pipeline involves attempting to parse the cleaned string into a Python dictionary using the `json.loads()` library. This validation acts as the first quality gate: if the model fails to output a valid JSON structure, the record is flagged as *Invalid*. This approach transforms raw, potentially messy autoregressive text into a pristine, structured format, ready to be

quantitatively assessed by the evaluation metrics framework described in the following section.

3.4.3 Natural Language Processing Evaluation Metrics Framework

The objective evaluation of the LLMs performance in extracting structured metadata from documents requires a robust mathematical framework. While the primary goal is the generation of valid JSON structures, the qualitative accuracy of the extracted information, such as component descriptions, materials, and technical specifications, must be measured against the human-annotated ground truth. To achieve this, the project implements a suite of **NLP metrics** that assess the similarity between the model's output, the *prediction*, and the gold standard, the *reference*. These metrics, ranging from *exact n-gram matching* to *semantic alignment*, provide various points of view of the extraction quality across the twelve experimental configurations.

To execute this evaluation at scale, the `inference.py` script iterates over the generated responses and invokes a centralized evaluation function for each extracted field, accumulating the results for the final analysis, as shown in code 3.24.

Code 3.24: Iteration over extracted fields and accumulation of NLP metrics

```
# NLP metrics for each extracted field
sample_field_metrics = {}
for key in global_accumulator.keys():
    m = f.compute_single_field_metrics(
        predicted_json.get(key, ""),
        target_json.get(key, "")
    )
    sample_field_metrics[key] = m
    for mn, mv in m.items(): global_accumulator[key][mn].append(mv)
```

The **BLEU** metric is implemented to evaluate the lexical precision of the models. BLEU calculates the geometric mean of modified n-gram precision between the candidate and reference texts, adjusted by a brevity penalty to prevent the overvaluation of short, incomplete responses [13]. It was chosen for this project because it provides a reliable measure of how many technical terms and sequences generated by the model exactly match the ground truth. Since individual JSON fields can often be very short, a variant known as **BLEU with smoothing** was also integrated. This technique addresses the issue of zero-precision scores for higher-order n-grams in short sequences by adding a small positive value to the counts, ensuring that even partial matches contribute to the

final score [3].

Code 3.25: Implementation of BLEU and smoothed BLEU metrics

```
# BLEU
b_std = bleu_metric.compute(
    predictions=[pred_str],
    references=[[ref_str]],
    max_order=max_ord
)
metrics["BLEU"] = float(b_std.get('bleu', 0.0))

# BLEU with smoothing
b_smooth = bleu_metric.compute(
    predictions=[pred_str],
    references=[[ref_str]],
    max_order=max_ord,
    smooth=True
)
metrics["BLEU-SMOOTHING"] = float(b_smooth.get('bleu', 0.0))
```

The **ROUGE** framework, and its variant **ROUGE-L**, is employed to assess the recall of the extraction process. ROUGE measures the overlap of n-grams between the computer-generated output and the reference, focusing on how much of the original information is successfully captured by the model [12]. ROUGE-L, in particular, utilizes the *Longest Common Subsequence* to identify the longest sequence of words appearing in both texts in the same relative order, which is particularly effective for evaluating the structural integrity of complex descriptions [12]. This metric ensures that no details are omitted during the automated processing.

Code 3.26: Implementation of ROUGE and ROUGE-L metrics

```
r = rouge_metric.compute(predictions=[pred_str], references=[ref_str])
metrics["ROUGE"] = float(r.get('rouge1', 0.0))
metrics["ROUGE-L"] = float(r.get('rougeL', 0.0))
```

The **METEOR** metric is included to bridge the gap between simple lexical matching and semantic similarity. METEOR improves upon BLEU by incorporating a generalized unigram matching process that accounts for stemmed forms and synonyms. It calculates a score based on the harmonic mean of unigram precision and recall, with a specific penalty for fragmentation to reward correct word ordering [1]. In the context of this project, METEOR is essential for jargon management, allowing it to find matches even for words that are not the same but are semantically

equivalent.

Code 3.27: Implementation of the METEOR metric

```
m = meteor_metric.compute(predictions=[pred_str], references=[ref_str])
valore_meteor = float(m.get('meteor', 0.0))
```

The technical execution of these metrics utilizes the *Hugging Face* `evaluate` library, centralized within the `inference_function.py` script. To ensure the interpretability and comparability of the results, the raw scores returned by these libraries, which typically range from 0 to 1, are systematically transformed into **percentages**. This conversion is handled during the generation of the final technical report, where each score is multiplied by 100. This choice was made to align the NLP metrics with standard reporting formats and give them greater interpretability, allowing for a more intuitive visualization of the performance gaps between the LoRA and QLoRA modalities. The final output provides the definitive benchmarks that will be analyzed in the final results chapter of this thesis.

4. Experimental Results and Discussion

The previous chapter **Project Methodology and Implementation** detailed the complete methodology of the project, starting from the automated acquisition and filtering of PDF documents, proceeding with the creation of a structured JSONL dataset and ending with the description of the training process and inference functions. The architectural choices behind the fine-tuning process on the HPC cluster were also explored, defining the parameter-efficient techniques and the inference framework required to test the models in a reproducible environment.

Building upon that technical foundation, this chapter presents the empirical results of the twelve training experiments conducted during this research. The primary objective is to move beyond the theoretical setup and analyze the actual performance of the chosen LLMs when tasked with extracting mechanical data. Through the use of NLP evaluation metrics and explanatory visual graphs, an analysis will be made of how well these models have learned the specific language of technical drawings.

The chapter is structured to provide a comprehensive and step-by-step evaluation of the entire experimental phase. First, section **Training Dynamics and Computational Efficiency** focuses on the training dynamics and computational efficiency. The validation loss will be analyzed to verify proper model convergence, ensuring that the models learned the underlying logic rather than merely memorizing the training data. Additionally, hardware utilization and the actual time required to train these architectures on the available GPUs will be evaluated.

Subsequently, section **Evaluation of Metadata Extraction** examines the practical outputs generated during the inference phase. The structural robustness of the generated JSON formatting will be assessed, and the true accuracy of the metadata extraction will be evaluated using NLP metrics, specifically BLEU, BLEU with smoothing, ROUGE, ROUGE-L and METEOR.

Finally, section **Comparative Analysis** provides a broad comparative analysis to answer the core research questions of this thesis. The LoRA and QLoRA training modalities will be directly compared to determine the real cost and impact of 4-bit quantization on final accuracy. Furthermore, empirical scaling laws will be investigated by comparing the performance of smaller 7 and 8 billion parameter architectures against larger 14 billion parameter models, culminating in a definitive

comparison of the foundation models to identify the most efficient solution for technical data extraction.

4.1 Training Dynamics and Computational Efficiency

In this section, the behavioral patterns and computational requirements observed during the fine-tuning phase of the selected LLMs are investigated. The adaptation of neural network weights for a specialized domain involves a balance between optimization quality and resource consumption. Consequently, analyzing how these architectures behave throughout the training cycles and how they interact with the underlying hardware infrastructure is essential to assessing the overall viability of the data extraction pipeline.

The evaluation is structured into two main subsections to address both the algorithmic and physical dimensions of the experiments. First, in section **Validation Loss and Convergence Analysis**, the learning progression is examined through the analysis of loss trajectories. This evaluation is directed toward verifying proper model convergence and detecting potential boundaries of overfitting or underfitting across the twelve executed configurations.

Second, in section **Hardware Utilization and Training Time**, the focus shifts to the infrastructural performance on the high-performance computing environment. The total training duration and volatile memory allocation are meticulously evaluated. This computational analysis highlights the constraints and performance trade-offs encountered when scaling from 7B and 8B architectures to larger 14B models, providing an empirical foundation for the subsequent comparative assessments.

4.1.1 Validation Loss and Convergence Analysis

The evaluation of training dynamics constitutes a fundamental phase in verifying the efficacy of the parameter-efficient adaptation process. Fine-tuning LLMs to perform structured metadata extraction requires continuous monitoring of cross-entropy loss functions to ensure proper alignment with the target serialization format without encountering overfitting phenomena. To maintain consistency across all evaluations, a **training dataset** composed of approximately 14,000 samples and a **validation dataset** composed of approximately 3,000 samples were utilized. Each model was trained over a duration of three complete epochs, with a maximum sequence length restricted

to 2,048 tokens to properly encapsulate the extensive context required by the structured output serialization. Furthermore, generalization capabilities were assessed by computing the validation metrics at a fixed frequency of 50 steps.

The optimization process was tracked over these iteration steps within the Unimore HPC cluster environment utilizing a single NVIDIA A40 GPU. Twelve distinct experimental configurations were executed to establish a direct comparative framework between LoRA and QLoRA across six separate model architectures. To ensure an equitable baseline for this hardware and software comparison, the structural adaptation parameters were maintained universally constant across all trials. Specifically, a **rank** (r) of 16, a **scaling factor** (α) of 32, and a **dropout probability** of 0.05 were applied. These specific values were selected to guarantee sufficient representational capacity for learning the mechanical taxonomy, while simultaneously preventing structural divergence and mitigating excessive computational overhead [9]. The remaining variable hyperparameter distributions and the resulting temporal metrics recorded during execution are summarized in table 4.1.

Table 4.1: Summary of the training hyperparameters and execution times applied across the experimental configurations.

Model	**Size**	**Mode**	**Time (h)**	**Learning Rate**	**Batch Size**	**Gradient Accumulation**	**Epochs**
Mistral	7B	lora	16.96	0.0002	4	8	3
Mistral	7B	qlora	22.68	0.0002	4	8	3
Phi3	8B	lora	13.28	0.0002	4	8	3
Phi3	8B	qlora	18.39	0.0002	4	8	3
Llama3.1	8B	lora	14.38	0.0002	4	8	3
Llama3.1	8B	qlora	19.41	0.0002	4	8	3
Qwen3	8B	lora	12.89	0.0002	4	8	3
Qwen3	8B	qlora	22.53	0.0002	4	8	3
Phi4	14B	lora	11.29	0.0001	1	32	3
Phi4	14B	qlora	13.14	0.0001	1	32	3
Qwen2.5	14B	lora	14.05	0.0001	1	32	3
Qwen2.5	14B	qlora	15.21	0.0001	1	32	3

During the initial phase of the training trajectory, a rapid and pronounced decline in the cross-

entropy training loss is observed uniformly across all language model families. This mathematical behavior indicates that the adapter layers successfully internalize both the syntactic conventions of the required JSONL schema and the semantic taxonomy describing components [7, 23]. To ensure objective validation monitoring, generalization capabilities were tested periodically at a validation frequency of 50 steps. The close alignment observed between the training loss curves and the corresponding validation loss records suggests a stable optimization path. This trajectory validates the configuration of the rank and scaling factor, which provided sufficient representation capacity for technical data extraction without introducing memory-type memorization or structural divergence.

To provide a macro-level understanding of the optimization trajectories, the global training loss behavior across all twelve experimental runs is illustrated in figure 4.1. Regardless of the underlying architecture or quantization constraints, a cohesive convergence pattern is observed, characterized by an initial steep descent followed by a stabilization phase. To isolate the effects of the tuning methodology, these global trajectories were subsequently decoupled into modality-specific visualizations.

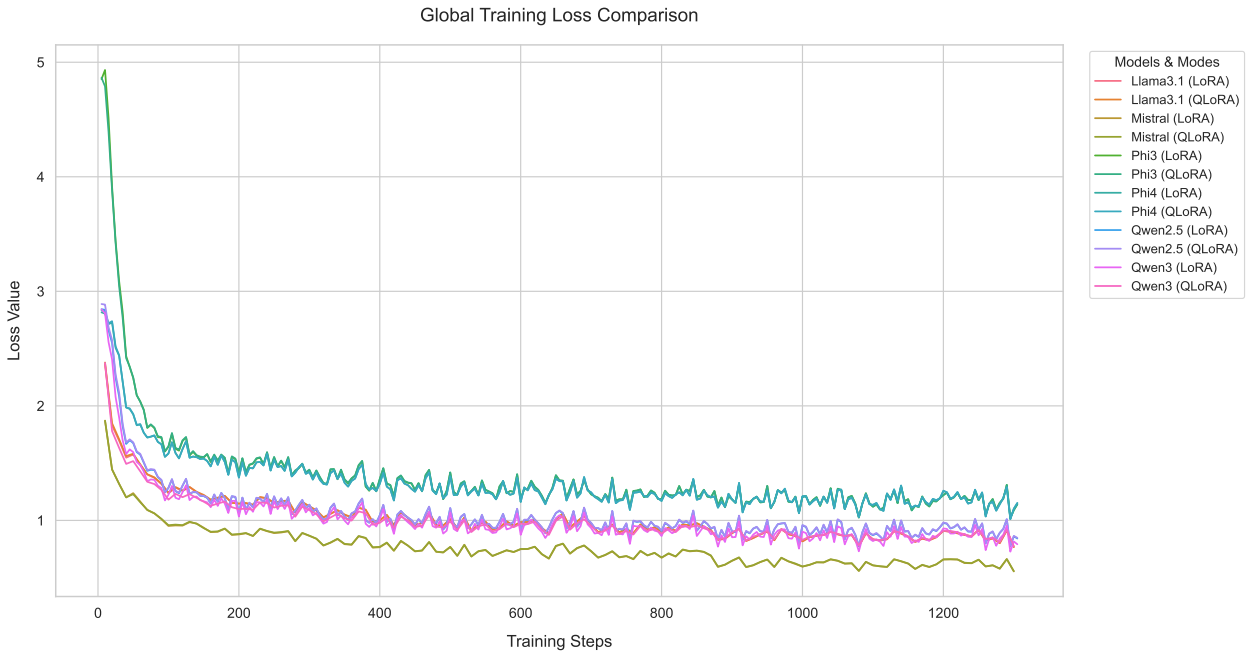

Figure 4.1: Global training loss convergence trajectories aggregated across all evaluated LLM and fine-tuning configurations.

The isolated convergence behavior for the full-precision adaptation is depicted in figure 4.2.

Under standard LoRA conditions, a uniform gradient descent is maintained across the model variants, with larger architectures demonstrating slightly accelerated initial stabilization.

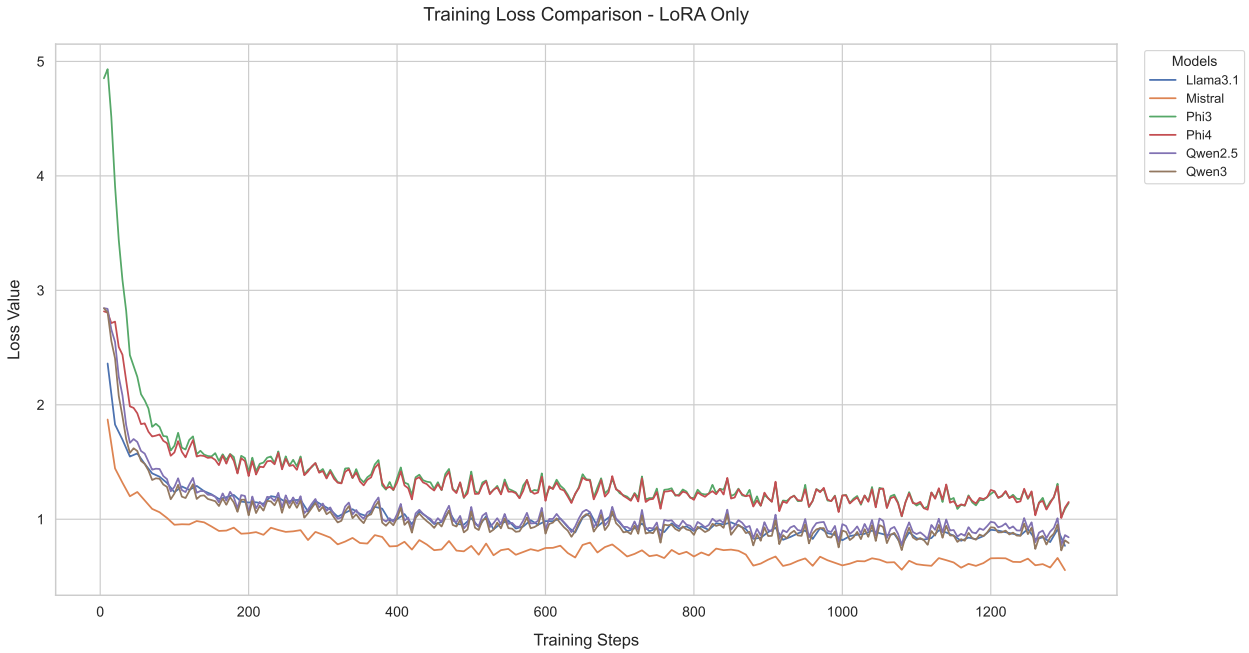

Figure 4.2: Loss trajectories specifically for the LoRA fine-tuning modality.

Conversely, the corresponding loss curves for the quantized optimization are detailed in figure 4.3. While the overall downward trend mirrors the full-precision counterpart, minor perturbations and slightly elevated variance during the intermediate epochs are evident in the QLoRA trajectories. This mathematical noise is a direct consequence of the 4-bit precision constraints during weight updates [4], although it does not ultimately prevent the architectures from reaching a stable convergence asymptote.

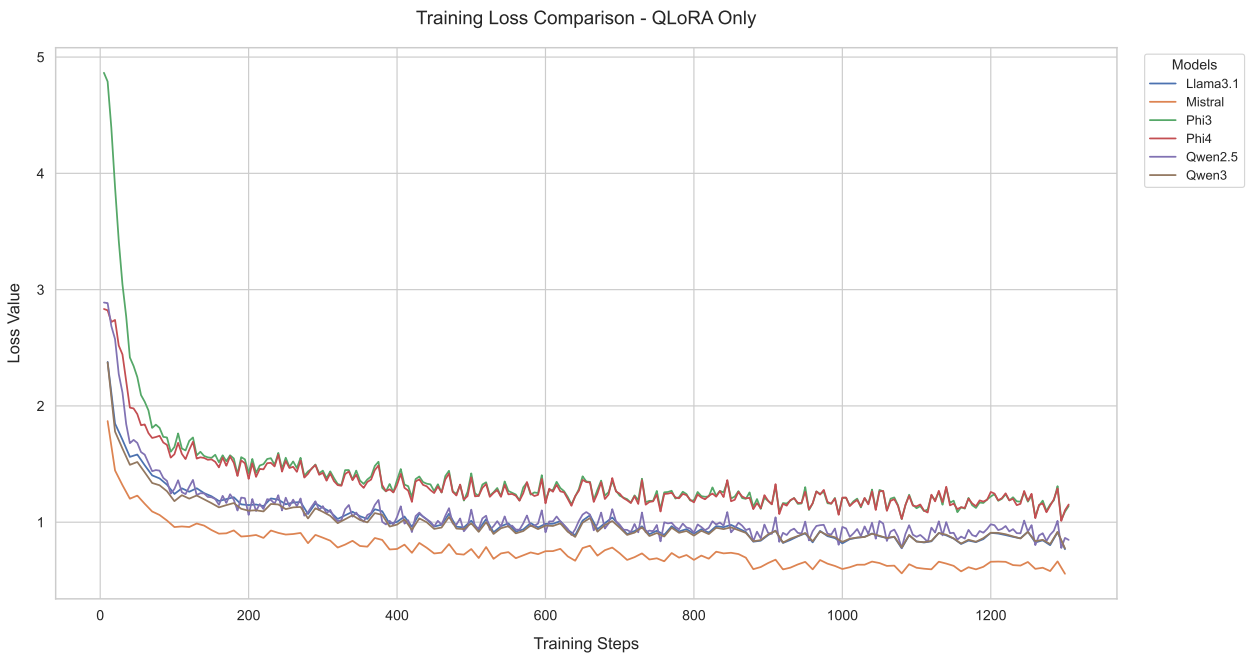

Figure 4.3: Loss trajectories specifically for the QLoRA fine-tuning modality.

A comprehensive perspective regarding the convergence behavior of the LLMs is provided through the exact loss trajectories of the most performant architecture, Mistral. Based on the empirical data extracted from the validation records, this architecture was identified as the most highly optimized configuration for the specific metadata extraction task. Superior adaptation capabilities were demonstrated by this model when compared to the larger 14B parameter variations. It is suggested by this performance delta that the baseline pre-training and instruction tuning of Mistral are exceptionally well-aligned with the robust structural decoding required to process texts affected by optical character recognition imperfections.

The detailed convergence tracking and loss behavior for the standard LoRA configuration are illustrated in figure 4.4. Under this modality, the lowest absolute validation loss among all evaluated models was achieved, reaching a final value of **0.7146**. A smooth tracking of the primary training metric by the validation loss was observed, with asymptotic stabilization occurring after the second epoch, while controlled variances were maintained by the gradient norms.

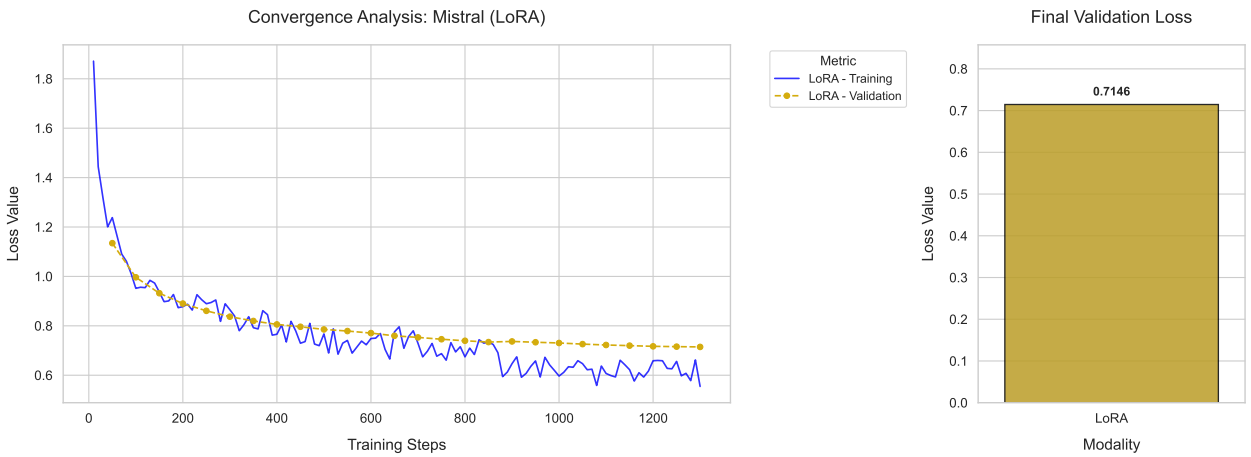

Figure 4.4: Detailed convergence tracking and loss behavior for the Mistral architecture under the LoRA configuration.

Subsequently, the training dynamics under quantized QLoRA constraints are detailed in figure 4.5. A competitive final validation loss of **0.7158** was reached by this configuration. It is highlighted by the visual comparison between the two learning curves that, although a temporal execution overhead is introduced by QLoRA, a nearly identical final convergence trajectory is achieved compared to its full-precision LoRA counterpart. Consequently, the structural integrity of the 4-bit quantization strategy is confirmed without compromising the final information extraction capabilities.

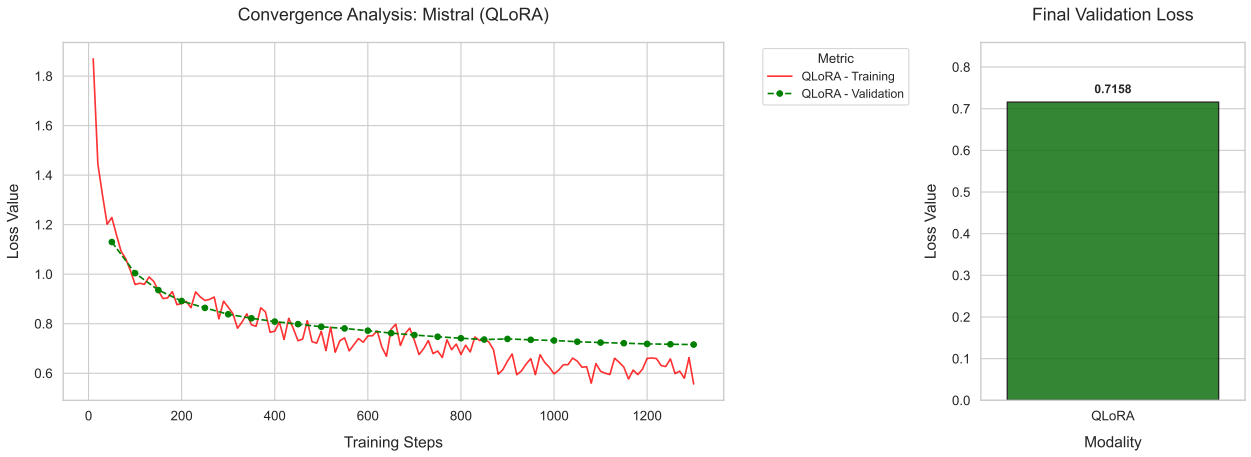

Figure 4.5: Detailed convergence tracking and loss behavior for the Mistral architecture under the QLoRA configuration.

4.1.2 Hardware Utilization and Training Time

An examination of computational efficiency and hardware resource allocation is useful to evaluate the practical feasibility of deploying parameter-efficient fine-tuning methodologies. When LLMs are adapted with LoRA and QLoRA to extract data, the theoretical advantages of parameter reduction must be balanced against raw execution time and physical hardware constraints. The interaction between model architecture scale, parameter quantization precision, and hardware resource saturation was continuously monitored, while running on the Unimore HPC cluster, to identify operational bottlenecks and throughput limitations.

The definitive impact of parameter quantization on temporal efficiency is illustrated in figure 4.6, where a direct comparative analysis of total execution times across different architectures and tuning modes is provided. A clear, systemic computational overhead is observed when transitioning from standard full-precision LoRA to quantized QLoRA execution.

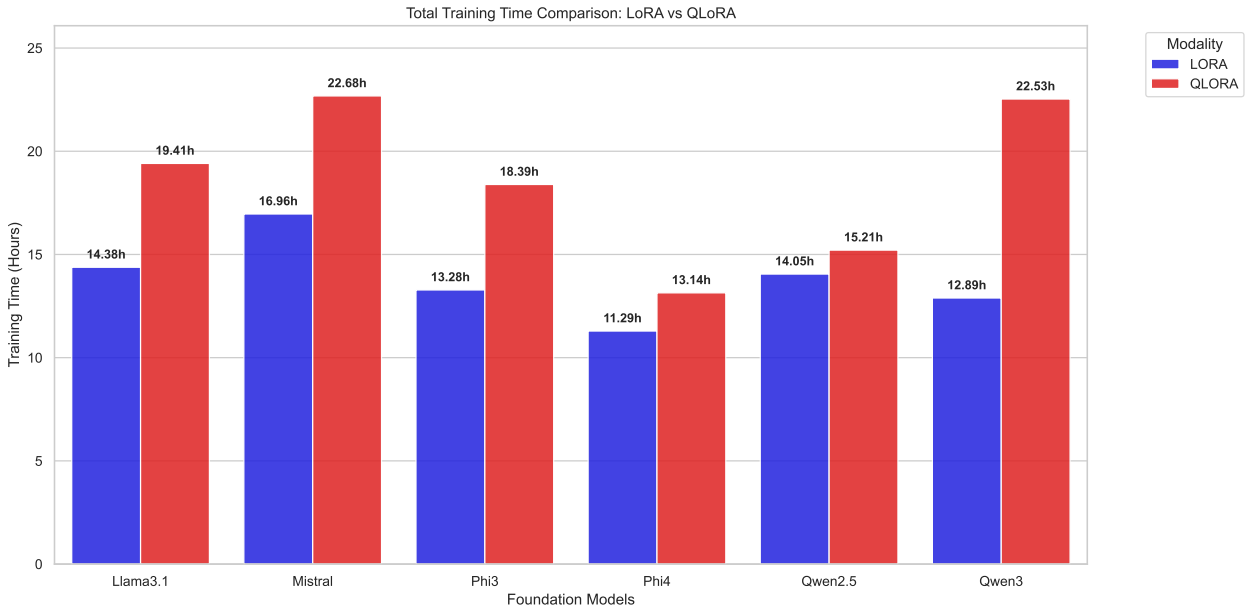

Figure 4.6: Comparative analysis of total training duration in hours across the selected LLM families under LoRA and QLoRA modalities.

This temporal penalty is directly mathematically attributed to the continuous **quantization-dequantization loops** executed during the forward and backward propagation passes. Because the base model weights are stored in a compressed NF4 representation to conserve memory, they

must be dynamically uncompressed into BF16 precision whenever a matrix multiplication step is performed by the tensor cores, before being compressed back [4].

This operational behavior is particularly evident in the 8B parameter family. For instance, the fine-tuning duration for the Qwen3 architecture exhibits a substantial increase from 12.89 hours in standard LoRA mode up to 22.53 hours under QLoRA constraints, representing a temporal expansion of approximately 74%. Similarly, the Mistral architecture demands 22.68 hours to complete the three training epochs in quantized mode compared to the 16.96 hours required by its full-precision counterpart. However, a significant anomaly in this scaling trend is encountered when analyzing the larger 14B parameter models, such as Phi4 and Qwen2.5. For these architectures, the temporal delta between LoRA and QLoRA is remarkably compressed. This behavior is explained by the strict memory mitigation strategies required to avoid out-of-memory errors on the physical hardware infrastructure.

To properly interpret these scaling dynamics, the relationship between model size and computational throughput must be evaluated, as visually detailed in figure 4.7. For the 7B and 8B architectures, the lower baseline memory footprint allows the allocation of a larger per-device batch size of 4, coupled with a gradient accumulation factor of 8. This configuration achieves an optimal level of parallelization, maximizing the utilization of the available hardware execution threads and resulting in elevated token-per-second processing throughput. Conversely, the 14B architectures introduce extreme memory pressure on the 48GB VRAM capacity. To maintain operational stability and prevent immediate memory exhaustion, the per-device batch size must be throttled down to a minimum value of 1, while the gradient accumulation steps are increased to 32 to ensure an identical effective batch size across all experiments [9].

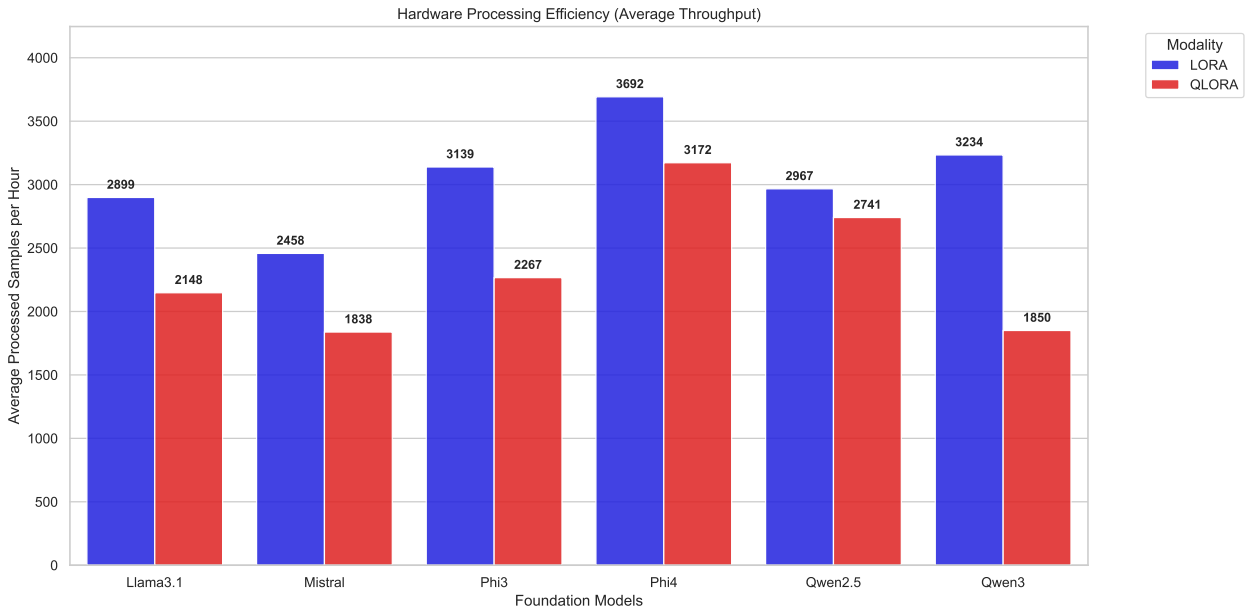

Figure 4.7: Average processed samples per hour across the selected LLM families under LoRA and QLoRA modalities.

The reduction of the micro-batch size to a single sequence severely limits data parallelization at the hardware level, shifting the primary computational bottleneck from the quantization-dequantization overhead to sequential memory access and gradient accumulation delays. Consequently, as demonstrated by the close execution profiles of the 14B variants, the memory savings achieved by QLoRA do not further degrade the temporal efficiency when the hardware is already heavily constrained by serialized execution bottlenecks. The selection between LoRA and QLoRA for large-scale architectures is therefore driven almost exclusively by hardware availability rather than execution speed trade-offs, as quantized mechanisms allow the deployment of superior baseline modeling capacities within memory boundaries that would otherwise remain physically inaccessible [4, 9].

4.2 Evaluation of Metadata Extraction

Following the assessment of training dynamics and computational resource utilization, this section shifts the focus toward the functional performance of the fine-tuned models. The primary objective of the proposed methodology is to transform unstructured technical documents into structured,

machine-readable formats. Therefore, the evaluation of metadata extraction capabilities is a critical step to validate the efficacy of the fine-tuning process.

The evaluation framework presented herein is designed to measure how effectively the LLMs generalize the instructions provided during the training phase. This assessment is bifurcated into two distinct perspectives to ensure a comprehensive understanding of the model's performance.

In section **Structural Robustness and JSON Formatting**, the analysis focuses on the model's ability to adhere to the defined schema. The syntactic correctness of the generated outputs is evaluated, ensuring that the extracted data conforms to the required JSON structure, which is essential for downstream integration into technical databases.

Subsequently, section **Natural Language Processing Metrics Analysis** provides a quantitative evaluation of the semantic accuracy and information extraction quality. By employing established NLP metrics, the generated extractions are compared against the ground truth dataset to quantify the model's proficiency in accurately identifying, classifying, and extracting mechanical component metadata.

By combining structural validation with semantic metric analysis, this chapter provides a comprehensive view of the extraction performance, highlighting both the successes and the persistent challenges encountered in the automated parsing of complex technical documentation.

4.2.1 Structural Robustness and JSON Formatting

The fundamental premise of automating data extraction lies in the capability to generate outputs that are not only semantically accurate but also syntactically rigid. To successfully integrate the extracted metadata into downstream ERP or product lifecycle management databases, the LLMs must adhere to a predefined serialization format without necessitating human intervention. In this specific architecture, JSON was selected as the target data-interchange standard due to its ubiquitous application and machine-readable nature.

One of the most persistent challenges encountered when deploying instruct-tuned LLMs for deterministic data extraction is their inherent tendency to generate conversational filler, formatting artifacts, or explanatory text surrounding the requested data. To enforce absolute format compliance, a restrictive system prompt was engineered and prepended to every inference request, so as to adapt pre-trained models to domain-specific formatting tasks [23, 7]. The models were explicitly mandated

to act as "expert mechanical engineers" and to return the metadata exactly within a valid JSON object, with extra text, explanations, or markdown formatting being expressly forbidden.

To quantitatively evaluate the success of this formatting constraint, a massive global inference campaign was conducted. The structural robustness was measured across 43,320 total inference samples, distributed equally among the twelve experimental configurations, 3,610 samples per model adaptation. A generated output was mathematically classified as valid exclusively if the resulting string could be parsed by a standard JSON decoder without raising any syntax exceptions.

The empirical evaluation of the generated outputs demonstrated an absolute level of structural robustness: an **exact format compliance rate of 100.0%** was achieved across all evaluated language models, regardless of whether the LoRA or the QLoRA modality was employed. This result validates the core principles of parameter-efficient fine-tuning methodologies. It is demonstrated that low-rank adaptation provides sufficient parameter updates to fundamentally alter the output distribution of a language model, successfully suppressing its conversational nature and aligning its generation capabilities directly with the requested deterministic schema [9]. Furthermore, the complete absence of malformed syntaxes under QLoRA configurations confirms that the 4-bit quantization constraints applied to the base weights do not degrade the structural integrity or the attention mechanisms required to consistently close braces and quotation marks over long generation sequences [4].

Consequently, the extraction pipeline resulted in consistently parsable JSON strings without the need for complex, regular-expression-based post-processing algorithms or extensive error-handling mechanisms for syntactic breakages.

The internal taxonomy of the extracted data was standardized across a fixed dictionary of six core attributes, which defined the precise target schema populated by the models. Specifically, the generated objects were required to include the alphanumeric identifier of the mechanical part, denoted by the key `DISEGNO N`, alongside a comprehensive textual synthesis of its characteristics, mapped to the `DESCRIZIONE` field. The categorical classification of the part was extracted into the `TIPO_COMPONENTE` attribute, whereas the specific raw material and its initial provisioning strategy were respectively serialized under `MATERIALE` and `TAGLIO MATERIALE`. Finally, the sequential list of subtractive, additive, or surface treatment operations was rigorously structured within the `LAVORAZIONI` key.

By internalizing this rigid schema mapping, the evaluated architectures demonstrated an exceptional capability to operate as deterministic data serialization engines. Since syntactic breakages and structural hallucinations were entirely eliminated across all the thousands of tested samples, it is established that any recorded discrepancies between the model outputs and the ground truth are strictly related to semantic understanding and contextual reasoning limits. This absolute structural reliability provides a stable and verified foundation for the quantitative assessment of information extraction quality, which is thoroughly investigated through NLP metrics in the subsequent section.

4.2.2 Natural Language Processing Metrics Analysis

Having established the structural reliability of the extraction pipeline, the evaluation focus is shifted toward the semantic accuracy of the generated content. To quantitatively assess the effectiveness with which the technical documents were interpreted and the required fields were populated by the LLMs, a suite of standard NLP metrics was employed. Specifically, BLEU, BLEU with smoothing, ROUGE, ROUGE-L and METEOR were calculated by comparing the model outputs against the human-verified ground truth. A robust mathematical framework to evaluate the precise information retention capabilities of instruction-tuned architectures is provided by the application of these lexical and n-gram overlap metrics [7, 23].

A global comparative overview of the average semantic performance across all evaluated architectures is visually summarized in figure 4.8. It is revealed by the aggregated metric analysis that the highest overall semantic extraction scores were achieved by the Llama 3.1 architecture, operating under the LoRA configuration, with a composite average exceeding 62.5% being registered across all evaluation criteria. Superior contextual understanding was demonstrated by this model, closely followed by the Mistral architecture, by which a highly competitive performance profile was exhibited. Interestingly, slightly lower lexical overlap scores were recorded by the larger 14B parameter variations, despite their theoretical capacity advantages. It is suggested by this behavior that the 8B architectures were more optimally aligned with the specific mechanical taxonomy during the fine-tuning phase.

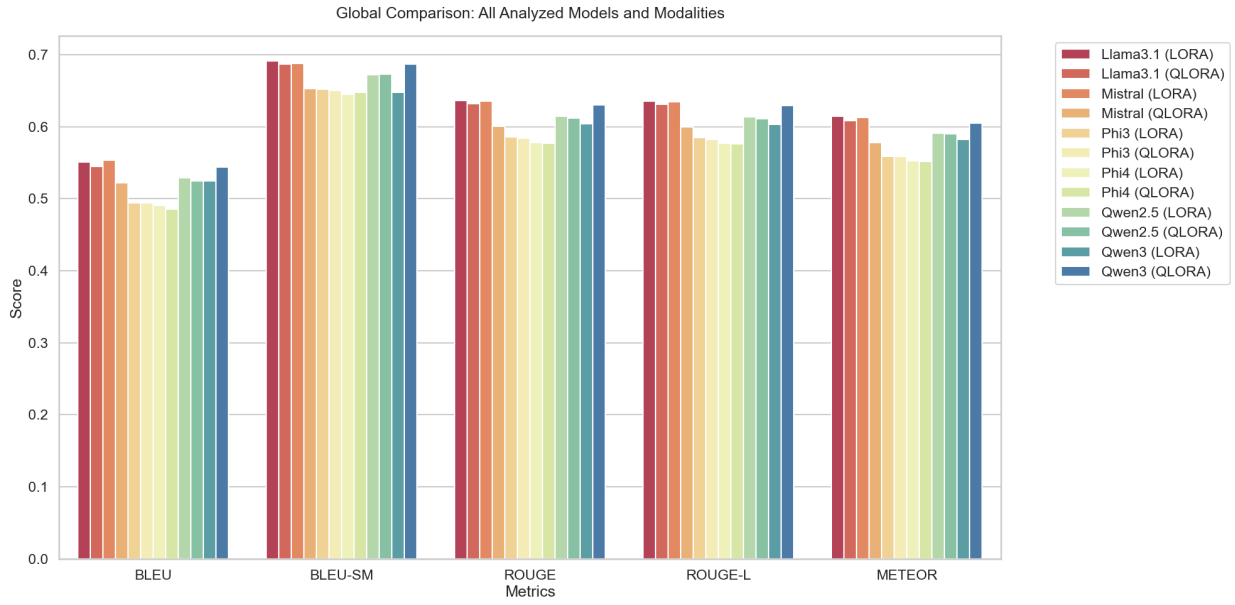

Figure 4.8: Global comparison of average NLP metric scores across all evaluated LLMs.

The impact of reduced precision on the semantic quality of the extraction is further investigated in figure 4.9, where the performance delta between standard adaptation and quantized constraints is detailed. A remarkably minimal degradation in semantic accuracy is observed when transitioning from LoRA to QLoRA. The aggregate metric scores are observed to experience a marginal decrease, generally confined within a **1% to 2%** variance margin. It is confirmed by this evidence that the core semantic reasoning capabilities of the base models are preserved by the NF4 quantization methodology [4]. Consequently, the extreme memory efficiency granted by quantization can be leveraged without significantly compromising the final quality of the extracted metadata [9].

To visually articulate this phenomenon, the average composite score for each evaluated architecture is plotted across both adaptation modalities. The resulting dual-bar trajectory provides a direct comparative baseline, clearly illustrating the nearly parallel performance trends and definitively highlighting the minimal gap between the standard and quantized precision regimes.

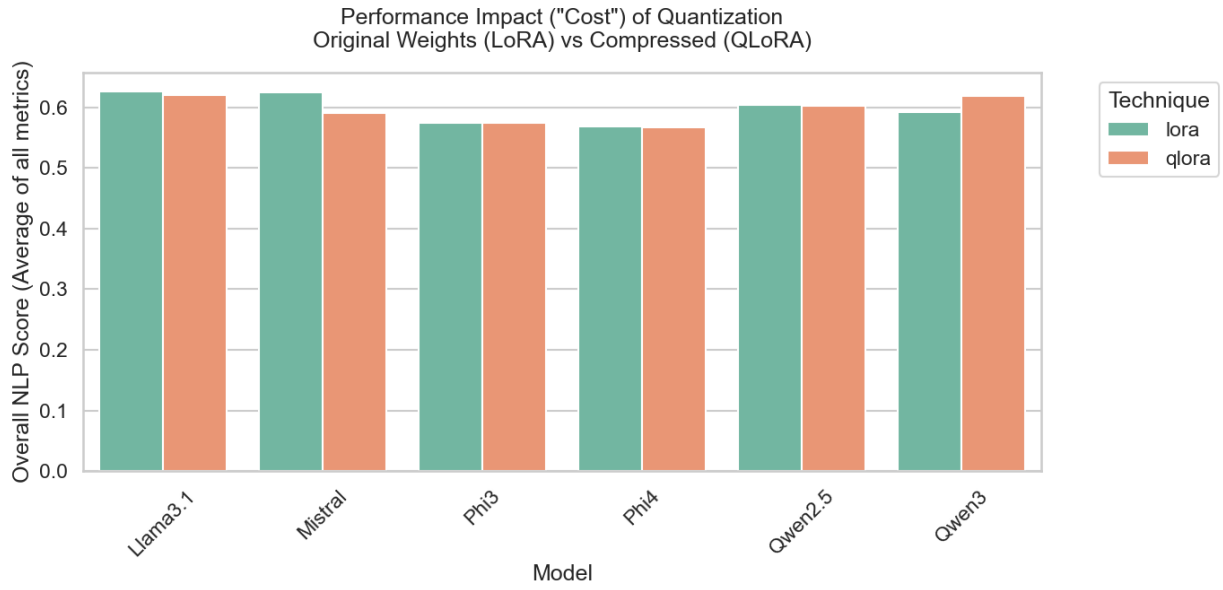

Figure 4.9: Analysis of semantic performance differences introduced by the QLoRA methodology.

To provide a comprehensive multidimensional perspective on the highest-performing configuration, the metric distribution for the absolute best overall model, the Llama 3.1 8B LoRA architecture, is illustrated through the radar chart in figure 4.10. A balanced proficiency across all dimensions is demonstrated, with higher overlap scores being consistently exhibited by BLEU with smoothing and ROUGE-L compared to the standard BLEU formulation. It is indicated by this discrepancy that while exact word-for-word matches are occasionally missed, the underlying structural sequence and the semantic core of the technical descriptions are accurately retained by the model.

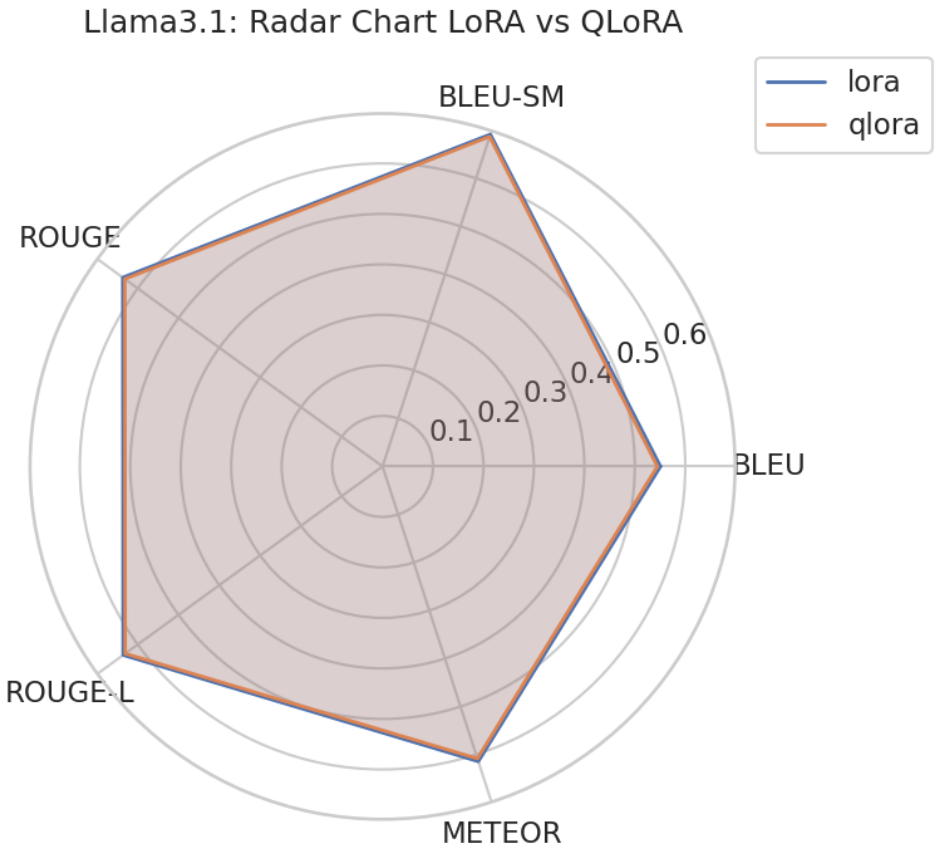

Figure 4.10: Radar chart detailing the metric performance of the Llama 3.1 LoRA architecture.

To further dissect the specific extraction complexities associated with the individual JSON fields across all configurations, a global performance heatmap is presented in figure 4.11. By analyzing the average metric scores distributed across the target attributes, a profound statistical dichotomy between the two leading architectures is revealed. While Llama 3.1 secured the highest global arithmetic mean, a superior consistency across the taxonomy was demonstrated by Mistral under LoRA constraints. Specifically, the highest absolute scores in three out of the six discrete fields were registered by Mistral: the open-text `DESCRIZIONE`, with 40.76% of accuracy compared to Llama's 40.44%, the alphanumeric `DISEGNO N`, with 65.57% of accuracy vs 64.58%, and `MATERIALE`, with 62.26% of accuracy vs 61.43%.

Conversely, Llama 3.1 achieved its victories with a significantly wider margin of perfor-

mance, particularly in standardized categorical fields. This architecture established dominance in LAVORAZIONI, with 64.58%, TAGLIO MATERIALE, with 73.92%, and notably TIPO_COMPONENTE, where a peak score of 70.46% was reached, substantially distancing Mistral's 68.74%. The magnitude of the positive mathematical delta accumulated by Llama 3.1 in these specific attributes effectively offset Mistral's broader numerical distribution of victories, validating Llama's position as the global average leader.

Furthermore, the heatmap exposes an inherent division in extraction difficulty depending on the nature of the target attribute. Exceptional accuracy rates, often exceeding 70%, are recorded by categorical, single-word fields. Conversely, the lowest scores are consistently recorded by the DESCRIZIONE field across all architectures. The variability of technical language is perfectly illustrated by this phenomenon: the correct mechanical concepts and dimensions are successfully extracted, but synonyms, alternative abbreviations, or different token orderings are frequently employed by the model compared to the strict ground truth definitions, causing exact-match metrics to decline while the semantic core is preserved.

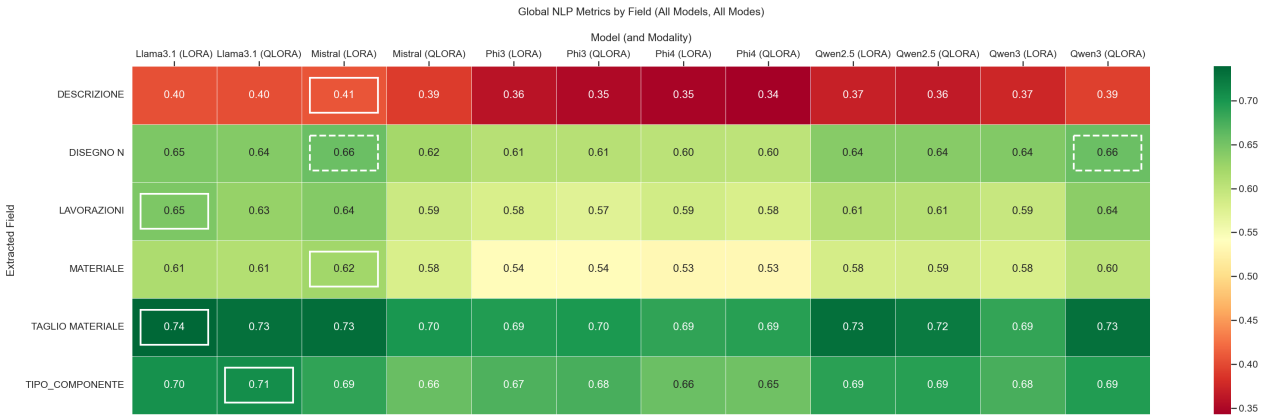

Figure 4.11: Global heatmap illustrating the average NLP metric performance across specific JSON fields for all evaluated model configurations.

By analyzing these aggregate results, it is established that the models are successfully transformed into capable extractors by the parameter-efficient adaptation. While rigid attributes are mapped with greater accuracy, semantic comprehension that exceeds the strict limitations of standard n-gram matching is demonstrated by the complex descriptive fields.

4.3 Comparative Analysis

Following the individual assessment of training dynamics, structural robustness, and semantic extraction accuracy, this section synthesizes the data to establish a comprehensive comparative framework. The primary objective is to translate the experimental findings into practical guidelines for the general application of parameter-efficient language models. By analyzing the tested architectures from different perspectives, the inherent trade-offs between computational demands, model size, and baseline performance are outlined.

The comparative investigation is articulated into three distinct dimensions. In section **The Cost of Quantization: LoRA vs. QLoRA**, the multifaceted impact of reduced-precision adaptation techniques is thoroughly examined. This analysis weighs the theoretical memory advantages and deployment feasibility of quantization against the practical temporal overhead introduced during the optimization phase and the corresponding semantic variations observed in the final metadata generation.

Subsequently, section **Scaling Laws: 7B/8B vs. 14B Architectures** investigates the correlation between the total parameter count and the specific adaptability to structured extraction tasks. The performance discrepancies between the compact 7B and 8B models and their larger 14B counterparts are examined. It is highlighted how physical memory limits and restricted batch sizes can surprisingly overturn standard scaling laws when the optimization is performed on constrained hardware.

4.3.1 The Cost of Quantization: LoRA vs. QLoRA

The integration of QLoRA configuration is widely celebrated for its ability to drastically reduce the VRAM footprint required during the fine-tuning of LLMs. By compressing the base model weights, memory bottlenecks are circumvented, allowing large architectures to be optimized on consumer-grade hardware [4]. However, the empirical data gathered during the experimental phase reveals that this extreme memory efficiency introduces a substantial computational trade-off, manifesting primarily as a severe temporal overhead during the training process.

By analyzing the execution logs documented in the training statistics, a consistent inflation in optimization time was recorded across all evaluated models when transitioning from standard LoRA

to QLoRA. For the Llama 3.1 architecture, the LoRA optimization over the dataset was concluded in 14.38 hours. Conversely, the exact same training regimen under QLoRA constraints required 19.41 hours, representing a temporal increase of approximately 35.0%. A nearly identical scaling behavior was observed for the Mistral model, where the optimization time spiked from 16.96 hours in LoRA to 22.68 hours under quantized conditions, translating to a 33.7% overhead. This computational delay is inherently caused by the continuous mathematical dequantization cycles required by the hardware: to perform the forward and backward passes, the compressed 4-bit weights must be temporarily cast back to 16-bit precision, saturating the GPU compute cores despite the reduced memory usage [4].

Despite this temporal tax, the quantitative evaluation of the convergence metrics indicates that the theoretical learning capacity of the architectures was not severely compromised by the reduced numerical precision. By examining the validation loss curves recorded at the terminal step of the training phase, the gap between the two modalities proved to be virtually negligible. The Llama 3.1 model achieved a final validation loss of 0.9170 under LoRA and 0.9193 under QLoRA. Similarly, the Mistral architecture reached an optimal validation loss of 0.7146 natively and 0.7158 when quantized. It is demonstrated by these figures that the essential feature representations and the structural distribution required to map the technical taxonomy were successfully preserved by the low-rank adapters, regardless of the base weight precision [9, 7].

However, as previously detailed in the NLP analysis in section **Natural Language Processing Metrics Analysis**, this identical mathematical convergence does not perfectly translate into identical generative accuracy. A marginal but consistent semantic degradation, bounded between 1% and 2% across lexical overlap scores, was registered for the quantized models.

While QLoRA remains a mandatory and highly effective strategy for deployment in strictly memory-constrained environments, standard LoRA optimization constitutes the superior architectural choice whenever sufficient hardware infrastructure is available. By avoiding base weight quantization, not only is the peak semantic extraction accuracy guaranteed, but the total computational time required for domain adaptation is reduced by over a third, resulting in significantly lower energy consumption and faster iteration cycles in pipelines.

4.3.2 Scaling Laws: 7B/8B vs. 14B Architectures

In the context of LLMs, scaling laws dictate that an increase in total parameter count theoretically yields proportionally superior semantic reasoning and zero-shot capabilities. To verify this paradigm within the specific domain of this project, a comparative evaluation was conducted between 7B and 8B architectures and larger 14B parameter models, specifically the Qwen2.5 and Phi-4 architectures. However, the empirical data extracted from the experimental phase reveals a paradoxical inversion of this theoretical scaling behavior when the fine-tuning process is subjected to hardware limitations.

During the optimization phase, the physical memory limits of the GPUs severely dictated the hyperparameters detailed in section **Training Hyperparameters**. To successfully fit the 14B models into the available VRAM alongside the LoRA and QLoRA adapter weights, the physical *batch size* per device was forcibly reduced to 1, compensated by an extended sequence of 32 gradient *accumulation steps*. In contrast, the 7B and 8B architectures operated with a physical *batch size* of 4 and 8 *accumulation steps*. While these configurations yield theoretically identical **effective batch sizes**, the extreme micro-batch reduction applied to the 14B models inevitably introduced severe noise into the gradient descent trajectory [9, 4]. Furthermore, deeper architectures are highly susceptible to *exploding gradients*: to maintain numerical stability during the initial adaptation phases and prevent this exponential error growth from destabilizing the weights, the *learning rate* for the 14B architectures was necessarily halved to 0.0001, keeping the total optimization runs to three epochs. This hardware-imposed reduction in both learning velocity degraded the stability of gradient updates, preventing the 14B parameter architectures from effectively mapping the formatting constraints without noise [7].

The detrimental impact of this forced hyperparameter configuration is quantitatively reflected in the terminal convergence metrics. Upon reaching the final training step, higher validation losses were recorded by the 14B models compared to their smaller counterparts. The Phi-4 architecture registered a validation loss of 1.1907 under LoRA constraints, which was higher than the 1.1214 loss achieved by the smaller Phi-3. Similarly, the Qwen2.5 14B model concluded the optimization with a validation loss of 0.9718, whereas the 8B version of the exact same evolutionary family achieved a more optimal 0.9365. Furthermore, the massive parameter count of the 14B models failed to bridge the gap with the overall leaders, trailing distantly behind the 0.7146 validation loss

natively recorded by the Mistral architecture.

This optimization deficit directly translated into inferior semantic extraction accuracy, as previously analyzed in the NLP metric evaluation in section **Natural Language Processing Metrics Analysis**. The 14B architectures plateaued at a global semantic overlap average of approximately 60.3% for Qwen2.5 and 56.8% for Phi-4, consistently underperforming both their direct 8B evolutionary predecessors. Consequently, it is established by this evidence that for deterministic data serialization tasks, sheer parameter scale is fundamentally less essential than the mathematical stability of the optimization process. When hardware constraints dictate sub-optimal batch sizes, PEFT adaptations applied to compact 8B architectures provide a vastly superior, stable, and accurate extraction framework compared to hardware-starved 14B models [23].

Beyond the scale-induced discrepancies, a final cross-architectural evaluation among the successful 7B and 8B baselines reveals distinct inherent adaptabilities to structured extraction tasks. While the Mistral architecture achieved the absolute lowest terminal validation loss with a value of 0.7146, the highest global semantic overlap of 62.57% was recorded by the Llama 3.1 model. It is suggested by this dichotomy that Llama 3.1 possesses a marginally superior semantic reasoning baseline, translating structural knowledge into more accurate lexical outputs [7]. Furthermore, a notable optimization anomaly was observed within the Qwen family: the Qwen3 model uniquely registered higher semantic scores under quantized QLoRA constraints, with 61.87%, than in LoRA, with 59.22%. This phenomenon implies that the NF4 compression acted as an effective regularizer, mitigating overfitting dynamics during the fine-tuning phase [4]. Conversely, the Microsoft Phi family exhibited the lowest structural compatibility, consistently recording the highest validation losses and lowest semantic accuracies, thereby demonstrating that their specific foundational baseline requires significantly more intensive adaptation to conform to rigid JSON constraints [23]. Ultimately, the Llama and Mistral architectures are established as the most robust foundational baselines for low-rank adaptation extraction pipelines.

5. Conclusions

Within the contemporary industrial landscape, the automated parsing of unstructured mechanical documents into deterministic data structures constitutes an engineering challenge. This research project was formulated to address this operational bottleneck by engineering a deterministic extraction pipeline based on LLMs. By leveraging PEFT techniques on the Unimore HPC cluster, the generative flexibility and probabilistic nature of these architectures were successfully constrained. The chosen models were transformed into rigorous, task-specific parsers capable of interpreting both native digital text and complex OCR artifacts. The primary objective of strictly enforcing a JSON serialization format was comprehensively achieved, effectively bridging the structural gap between chaotic technical drawings and machine-readable relational databases. Through the orchestration of automated data acquisition covering around twenty thousand documents, conversational instruction tuning, and inference strategies, a reliable bridge between visual documents and programmatic data processing was definitively established.

The absolute operational success of the engineered extraction pipeline was confirmed by the empirical evaluation conducted during the inference phase. Across a massive testing volume comprising thousands of unseen mechanical documents, an exact format compliance rate of one hundred percent was recorded by the adapted models. It was demonstrated that the conversational hallucinations typically associated with generative AI were successfully reduced to a minimum by the specific conversational triplet format, combined with a near-zero temperature setting. Consequently, the generated JSON outputs were consistently parsed without raising any syntactic exceptions. The main hypothesis of the research is confirmed by this total absence of formatting errors: the parameter updates introduced by LoRA are proven to be sufficient to force the model to respect the required JSON structure without deviations. A solid baseline for the automated categorization of mechanical metadata is established by completely avoiding syntax mistakes. Ultimately, it is demonstrated that LLMs can stop generating unpredictable text and work as reliable data extractors when constrained by precise instruction formats.

The true operational cost of QLoRA was clearly isolated during the training phase. By compressing the foundational weights into four bits, hardware VRAM limits were successfully bypassed,

allowing large architectures to be trained on standard computing resources. However, a significant slowdown was caused by this compression technique. A continuous decompression process back into sixteen-bit precision was required by the hardware to execute mathematical operations, and the total training time was subsequently increased by approximately 35%. Despite this temporal delay, it was verified that the final extraction accuracy remained almost identical. The performance drop measured by the NLP metrics, specifically BLEU, BLEU with smoothing, ROUGE, ROUGE-L and METEOR, was limited to a minimal 1% or 2% difference compared to standard LoRA. Therefore, it is concluded that standard LoRA is preferred when GPU memory is abundant, but an exceptionally solid solution for training models under strict hardware limitations is provided by QLoRA.

An unexpected result was observed regarding model size: the 14B parameter architectures performed significantly worse than the smaller 7B and 8B parameter models. While it is generally assumed that larger parameter counts yield better performance, the strict memory limits of the available GPUs completely reversed this trend. To prevent VRAM exhaustion, the training batch size for the larger models had to be restricted to a single example. This extreme limitation caused unstable parameter updates during the training phase. To manage this instability, the learning rate was halved, which ultimately prevented the models from learning effectively and reaching optimal convergence. Consequently, it is demonstrated that when hardware resources are constrained, a stable training loop with adequate batch sizes is far more essential than raw model capacity for rigorous JSON data extraction. The most accurate results are definitively provided by compact architectures trained under optimal configurations.

Among the evaluated models, Llama 3.1 and Mistral were proven to be the most effective solutions. The highest scores across all NLP metrics were achieved by Llama 3.1, which demonstrated precise reasoning for categorical classifications and single-word material extractions. Concurrently, the lowest validation loss was reached by Mistral. This specific architecture showed an exceptional ability to process long, open-text descriptions and perfectly adapt to the variable terminology of mechanical jargon. Therefore, these two foundational models are designated as the optimal choices for industrial data extraction pipelines. Building upon these results, future developments could explore the direct integration of multimodal vision-language models. By analyzing images directly, geometric features and spatial layouts could be extracted straight from the technical diagrams, completely bypassing the OCR phase. Finally, by increasing the context window limits, the simultaneous

processing of complex, multi-page technical assemblies could be enabled, further expanding the autonomy of the automated pipeline.

Bibliography

[1] Satanjeev Banerjee and Alon Lavie. “METEOR: An automatic metric for MT evaluation with improved correlation with human judgments”. In: *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 2005, pp. 65–72.

[2] Tom Brown et al. “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[3] Asli Celikyilmaz, Elizabeth Clark, and Jianfeng Gao. “Evaluation of text generation: A survey”. In: *arXiv preprint arXiv:2006.14799* (2020).

[4] Tim Dettmers et al. “Qlora: Efficient finetuning of quantized llms”. In: *Advances in neural information processing systems* 36 (2023), pp. 10088–10115.

[5] Jacob Devlin et al. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 2019, pp. 4171–4186.

[6] Alexey Dosovitskiy et al. “An image is worth 16x16 words: Transformers for image recognition at scale”. In: *arXiv preprint arXiv:2010.11929* (2020).

[7] Chang Gao et al. “Jsontuning: Towards generalizable, robust, and controllable instruction tuning”. In: *Findings of the Association for Computational Linguistics: ACL 2025*. 2025, pp. 24029–24055.

[8] Javier Hoffmann et al. “A survey on CNN and RNN implementations”. In: *PESARO 2017: The Seventh International Conference on Performance, Safety and Robustness in Complex Systems and Applications*. Vol. 3. 2017.

[9] Edward J Hu et al. “Lora: Low-rank adaptation of large language models.” In: *Iclr* 1.2 (2022), p. 3.

[10] Jared Kaplan et al. “Scaling laws for neural language models”. In: *arXiv preprint arXiv:2001.08361* (2020).

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).

[12] Chin-Yew Lin. “Rouge: A package for automatic evaluation of summaries”. In: *Text summarization branches out*. 2004, pp. 74–81.

[13] Kishore Papineni et al. “Bleu: a method for automatic evaluation of machine translation”. In: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, pp. 311–318.

[14] Nusrat Jahan Prottasha et al. “Peft a2z: parameter-efficient fine-tuning survey for large language and vision models”. In: *arXiv preprint arXiv:2504.14117* (2025).

[15] Alex Sherstinsky. “Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network”. In: *Physica d: Nonlinear phenomena* 404 (2020), p. 132306.

[16] Ray Smith. “An overview of the Tesseract OCR engine”. In: *Ninth international conference on document analysis and recognition (ICDAR 2007)*. Vol. 2. IEEE. 2007, pp. 629–633.

[17] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to sequence learning with neural networks”. In: *Advances in neural information processing systems* 27 (2014).

[18] Hugo Touvron et al. “Llama 2: Open foundation and fine-tuned chat models”. In: *arXiv preprint arXiv:2307.09288* (2023).

[19] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[20] Ramakrishna Vedantam, C Lawrence Zitnick, and Devi Parikh. “Cider: Consensus-based image description evaluation”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 4566–4575.

[21] Luping Wang et al. “Parameter-efficient fine-tuning in large language models: a survey of methodologies”. In: *Artificial Intelligence Review* 58.8 (2025), p. 227.

[22] Chaohao Yuan et al. “A survey of graph transformers: Architectures, theories and applications”. In: *arXiv preprint arXiv:2502.16533* (2025).

[23] Shengyu Zhang et al. “Instruction tuning for large language models: A survey”. In: *ACM Computing Surveys* 58.7 (2026), pp. 1–36.

[24] Yue Zhang et al. “Siren’s Song in the AI Ocean: A Survey on Hallucination in Large Language Models”. In: *Computational Linguistics* 51.4 (2025), pp. 1373–1418.