Compression and Optimization of LLMs for Low-Latency, CPU-Based Inference at Enterprise Scale

DISSERTATION

Submitted in partial fulfillment of the requirements of the
Degree : MTech in Data Science & Engineering

By

Chandan K S
2023DA04296

Under the supervision of

Harshit Gupta
Staff Data Scientist & Team Lead

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
Pilani (Rajasthan) INDIA

August, 2025

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to the Birla Institute of Technology and Science, Pilani (BITS Pilani) for providing me with this invaluable opportunity to pursue my M.Tech dissertation. I am especially thankful to the Work Integrated Learning Programme (WILP) and to **Prof. Pritee Agarwal** for her continuous support, guidance, and constructive feedback throughout the evaluation process. Her insights have greatly strengthened the quality of this work.

I would also like to extend my heartfelt thanks to **Mr. Vinit Bachhav**, my mentor at Truecaller, who has guided me not only through the course of this project but also throughout my M.Tech journey. His direction, encouragement, and mentorship were instrumental in shaping both the technical and research aspects of this dissertation. I am equally grateful to **Mr. Harshit Gupta**, my manager and additional examiner, for his support, valuable input, and perspective that helped me refine my approach to the project.

This journey has been both challenging and rewarding, and I am deeply appreciative of the constant motivation and support I received from my colleagues at **Truecaller**, which made this endeavor possible.

Lastly, I am profoundly grateful to my parents for their unconditional love, encouragement, and unwavering belief in me. Their support has been the foundation of my academic and professional growth.

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

CERTIFICATE

This is to certify that the Dissertation entitled **Compression and Optimization of LLMs for Low-Latency, CPU-Based Inference at Enterprise Scale** and submitted by Mr. **Chandan K S** ID No. **2023DA04296** partial fulfillment of the requirements of DSECLZG628T Dissertation, embodies the work done by him under my supervision.

Harshit

Signature of the Supervisor

Place: Bengaluru

Date: 14th August, 2025.

Name : Harshit Gupta

Designation : Staff Data Scientist & Team Lead

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
SECOND SEMESTER 2024-25

DSECLZG628T DISSERTATION

Dissertation Title : **Compression and Optimization of LLMs for Low-Latency, CPU-Based Inference at Enterprise Scale**

Name of Supervisor : **Harshit Gupta**

Name of Student : **Chandan K S**

ID No. of Student : **2023DA04296**

Courses Relevant for the Project & Corresponding Semester :

1. Mathematical Foundations for Data Science - 1st
2. Introduction to Data Science - 1st
3. Machine Learning - 2nd
4. Deep Learning - 3rd

Abstract

Large Language Models (LLMs) have shown remarkable capabilities across natural language tasks, but their deployment at scale remains a challenge due to high computational demands and latency-especially when GPUs are not viable. For many enterprise applications that require ultra-low latency and high throughput (QPS), such as fraud detection or real-time search ranking, general-purpose LLMs are too large and inefficient.

This dissertation aims to develop a compression and optimization pipeline to convert general-purpose LLMs into lightweight, task-specialized models that run efficiently on CPUs. The goal is to reduce model size and latency while maintaining task performance, enabling real-world enterprise deployment without GPU dependency.

The primary focus will be on quantization-aware fine-tuning and model pruning, which together can significantly reduce memory footprint and inference time. Techniques like knowledge distillation and CPU-specific optimizations (e.g., ONNX runtime tuning, multi-threaded inference) may also be explored depending on feasibility, but are not central to the core objective.

To evaluate this pipeline, the project will use a name matching task-where a model must classify whether two names refer to the same entity (match, no match, partial match). This is a common real-world task in domains like KYC (Know Your Customer), user identity linking, and database deduplication. A labeled name-pair dataset will serve as the evaluation benchmark to measure model accuracy, latency, and scalability.

The methodology involves:

- Selecting a compact LLM Model (e.g., LLaMA 1B or similar),
- Fine-tuning the model on the name matching dataset using parameter-efficient strategies (e.g., LoRA),
- Applying quantization-aware training to enable low-precision inference (e.g., INT8),
- Conducting structured/unstructured pruning to remove redundant weights,
- Benchmarking performance (F1 score, confusion matrix, inference latency),
- Optionally exploring further CPU-level optimizations and distillation if resources and time allow.

The expected outcome is a quantized and pruned model, ideally around 500 Million to 1B parameters , capable of fast CPU-only inference suitable for enterprise-scale deployments. More importantly, the project seeks to create a reusable, task-agnostic optimization pipeline that can be applied to other narrow use cases, making LLMs production-ready for constrained environments.

Key Words: *LLMs, quantization, pruning, CPU inference, low latency, high QPS, enterprise deployment, name matching, model compression, optimization pipeline*

List of Symbols & Abbreviations used

Abbreviation / Symbol	Expanded Meaning
AVX-512	512-bit Advanced Vector Extensions (Intel SIMD instruction set)
CPU	Central Processing Unit
F1	Macro-averaged F1 Score
FLOP / FLOPs	Floating-Point Operation(s)
GPU	Graphics Processing Unit
INT4	4-bit integer weight precision
INT8	8-bit integer weight precision
KYC	*Know Your Customer* use-case
LLC	Last-Level Cache (shared L3)
LLM	Large Language Model
LoRA	Low-Rank Adaptation
NF4	*Normal-Float-4* quantisation format
ONNX	Open Neural Network Exchange (inter-framework model format)
PEFT	Parameter-Efficient Fine-Tuning
QLoRA	Quantized Low-Rank Adaptation
QAT	Quantization-Aware Training
QPS	Queries Per Second
SLA	Service-Level Agreement (latency / uptime contract)
TCO	Total Cost of Ownership (hardware + energy + operations)
GGUF	General-purpose GPU/CPU Unified Format (model format for llama.cpp inference)
W&B	Weights & Biases (experiment tracking platform)
bnb	BitsAndBytes (quantization library)

List of Tables

List of Figures

Table of contents

Chapter 1

Introduction

1.1 Background

In consumer facing use cases where it is known the person is using an LLM, it is acceptable for an LLM to have delay. In enterprise production systems, LLM's will be used for various purposes like fraud-detection, search-ranking engines, sentiment analysis and entity matching. Where the consumer is not aware of the technology used, the latency expectation will be very low. And can be devastating in few cases like fraud-detection. Teams therefore impose strong SLA’s (service level agreements) to have fixed latency where if the tokens/second is less than the SLA then the product won’t work as expected and won’t be used.

GPUs excel at dense matrix math but are scarce, expensive and power-hungry in many corporate data-centres. Large banks, telcos and government clouds often restrict GPUs for regulatory or cost reasons; yet these organisations still hold vast CPU clusters [14]. A single 1 B-parameter model can fit inside the LLC of a modern CPU or EPYC socket once it is quantised to 4–8 bit-but only if the model is further compressed and sparsified so it can be executed with cache-resident weights and minimal memory bandwidth.

Moreover, GPU provisioning leads to bursty utilisation (idle for long periods, saturated during spikes) whereas CPUs are already present and can be elastically scheduled alongside existing micro-services. By running on CPUs we also avoid vendor lock-in and can deploy on edge devices or air-gapped servers where no GPU is available.

Thus, for cost, availability, energy and latency reasons, enterprises need a systematic way to transform generic LLMs into CPU-friendly, low-latency, task-specialised models without sacrificing critical accuracy.

1.2 Problem Statement

General-purpose foundation models such as GPT-3, Llama-2 & 3, and Mixtral are engineered for maximal breadth of capability, not deployment efficiency. A single 7–13 B-parameter checkpoint in fp16 occupies 14-26 GB of memory and demands >200 GFLOPs per generated token [15][16].

In practice this means:

1. Memory pressure. Even a modest 7 B-parameter model cannot fit into the L3 cache of a modern CPU; weights must stream from DRAM, creating latency spikes and wasting bandwidth that would otherwise serve business logic.

2. Compute saturation. Transformer inference is dominated by matrix-multiplication and scaled-dot-product attention. Without GPUs or specialised accelerators, a CPU must execute billions of multiply-adds per request, pushing response-time well beyond the sub-100 ms envelope that enterprise SLAs impose for synchronous services (e.g., fraud-scoring per credit-card swipe).

3. Cost & energy. Renting or installing datacentre GPUs increases TCO by \$10 K–\$15 K per device and draws ≈ 300 W continuously. In contrast, CPUs are already provisioned; utilising them avoids extra cap-ex and simplifies scheduling alongside existing micro-services.

Therefore organisations with strict latency, cost or regulatory constraints cannot simply drop a stock LLM into production.
They need a systematic pipeline that
(i) learns only the task-relevant subspace
(ii) compresses weights through quantisation and pruning
(iii) delivers an artefact that fits entirely in CPU cache and responds in tens, not hundreds, of milliseconds without sacrificing the decision-quality demanded by name-matching and similar entity-resolution tasks

1.3 Objectives & Scope

The central aim of this dissertation is to craft a re-usable optimisation pipeline that transforms a general-purpose large language model into a task-specialised, cache-resident engine that can answer queries in real time on commodity CPUs. All experimentation so far has been anchored to a concrete name-matching classification task in which the model must decide whether two person-name strings represent the same individual (yes), different individuals (no), or partially overlapping identities (partial). The corpus consists of 2404 labelled name pairs that were cleaned, de-duplicated, and class-balanced through stratified oversampling. Each example is formatted into a concise prompt that asks the model for a one-word answer, and evaluation is performed with a standard five-fold split to guard against random-seed luck.

Our primary evaluation metric is macro-averaged F1, chosen because it weights all three classes equally and highlights weaknesses in minority labels that overall accuracy might conceal. Alongside this, we track overall accuracy, per-class F1, and inference latency to expose hidden trade-offs that matter for real-time enterprise deployment.

Based on the completed experiments, the final pipeline has crystallised into three steps:

1. **Parameter-Efficient Fine-Tuning in fp16:** We first apply LoRA-based fine-tuning in full precision. This allows the adapters to explore the richest possible weight space and converge on a stable decision boundary. At this stage, the model achieves ≈ 87 % accuracy and ≈ 0.87 macro-F1, which becomes the upper benchmark for all subsequent compression.
2. **Quantisation with Recovery (INT8 → INT8 QLoRA) :** Post-training INT8 quantisation reduces memory footprint but drops accuracy to ≈ 81 %. A recovery pass with INT8 QLoRA then restores performance to ≈ 85 % accuracy and ≈ 0.836 macro-F1, while shrinking memory by ~3.7×. This “INT8 recovery path” is the sweet-spot, striking a balance between accuracy and low-latency CPU inference.
3. **Deployment via .gguf Conversion :** The optimised INT8 model is exported to GGUF format and served through llama.cpp, enabling cache-resident CPU inference.

Latency drops below enterprise SLA thresholds while completely avoiding GPU costs, making the pipeline practical for production.

Structured pruning was evaluated but ultimately removed from the pipeline. On a 1 B-parameter backbone, pruning reduces accuracy significantly without yielding meaningful latency gains—because the model is already small enough to fit into CPU cache once quantised. For this specific use-case (name matching), pruning only harmed performance. However, pruning may remain relevant for larger models (7–13 B) deployed on tasks that demand richer context windows, where slimming down depth or channels can yield both accuracy and throughput benefits.

In summary, the final CPU-ready pipeline is:

fp16 LoRA fine-tune → INT8 QLoRA recovery → GGUF conversion for CPU inference.

This reduces accuracy by only ~5 pp relative to the fp16 baseline while saving substantial cost, energy, and latency. A comparison table of accuracy, F1, memory footprint, and tokens-per-second at each stage will be included here.

1.1 List of objectives and status

	Objective	**status**	**comment**
1	Develop a reusable optimisation pipeline to compress LLMs for low-latency CPU inference	Done	End-to-end pipeline fully automated via Python scripts for data ingestion, fp16 LoRA fine-tuning, INT8 QLoRA recovery, and GGUF INT8 export. Deployable via llama.cpp for CPU inference, achieving sub-100 ms latency on AMD EPYC CPUs. ONNX/OpenVINO wrapper not pursued as GGUF meets enterprise SLA requirements.
2	Identify the best strategy to shrink model size for narrow tasks while preserving accuracy	Done	INT8 QLoRA with one recovery epoch delivers 0.836 macro-F1 (~2 pp below fp16 baseline) and ~3.7× smaller memory footprint (~1.2 GB). GGUF INT8 export achieves 0.82 macro-F1 with ~20 tokens/sec on CPU. Structured pruning (14/16 layers) was evaluated but excluded due to ~6 pp F1 loss outweighing marginal latency gains.
3	Experiment with sequencing of quantisation, pruning and	Done	Three sequences tested: (a) fp16 LoRA → INT8 (no recovery, 0.811 macro-F1), (b) fp16 LoRA → INT8QLoRA + recovery (0.836 macro-F1), (c) fp16 LoRA → pruning → INT8 (unviable due to

	(optionally) distillation / CPU tuning		accuracy loss). Pruning after QLoRA erased recovery gains. Knowledge distillation and ONNX/OpenVINO tuning not pursued as fp16 LoRA + QLoRA + GGUF INT8 meets performance and cost goals.
4	Evaluate optimised models on macro-F1, class F1, confusion matrix & latency	Done	Full metrics (macro-F1, per-class F1, accuracy, confusion matrix, tokens/sec, latency) tracked via W&B for 5-fold CV (0.857 ± 0.009 macro-F1) and 95%- train run (0.896 macro-F1). Latency audited on AMD EPYC (CPU), NVIDIA L4, and A100 GPUs. GGUF INT8 achieves ~20 tokens/sec and ~77 ms latency on CPU, scaling to 100 QPS with 5 VMs (~$1.5k/month), ~30× cheaper than fp16 GPU.

Chapter 2

Baseline Experiments & Quantisation

Out-of-the-Box 1 B Model Baseline Failure

Before any task-specific fine-tuning we briefly evaluated the vanilla 1 B-parameter Llama-3 checkpoint on the raw name-matching set using the short-prompt template introduced below. With no LoRA adapters and no gradient updates the model managed an overall accuracy of only 0.514 and a macro-F1 barely above chance (≈0.52). Manual inspection of the generated sequences revealed that instead of emitting a single classification token the model frequently drifted into generic completions such as "Here is the answer:" or attempted to justify its choice with multi-token sentences. Because evaluation truncates everything beyond the first token this behaviour translated into systematic label errors and depressed scores across all classes.

The experiment underscores two take-aways:

(i) small-footprint LLMs cannot solve the task without supervision.

(ii) instruction prompts must be paired with explicit single-token loss masking during training so the decoder learns to terminate immediately after producing yes, no or partial.

2.1 Dataset Preparation & fp16 Training Setup

The first empirical milestone was to establish a **full-precision LoRA baseline [4]** against which every subsequent compression step could be compared. Prior to any balancing the raw name-matching corpus was heavily skewed-≈60 % “yes”, 27 % “no”, and only 13 % “partial”. Such an imbalance would bias a naïve classifier toward over-predicting matches, so the minority classes were **oversampled with stratified bootstrap sampling** until all three labels were equally represented. The final balanced split comprises 2 404 prompt-response pairs, each rendered into a concise instruction template that asks the model to output exactly one token: *yes*, *no* or *partial*.

For training we wrapped the 1 B-parameter Llama-3 backbone with rank-16 LoRA adapters [4] per attention block and fed batches through a **completion-only data collator** that masks loss outside the answer span, ensuring gradient signal focuses on the decisive token. Five-fold stratified cross-validation was adopted to dampen variance from random seeds and to expose any fold-specific brittleness. During each run we stream **macro-F1, overall accuracy, per-class F1, validation loss and token-level perplexity** to Weights & Biases; the dashboard provides real-time curves plus confusion-matrix snapshots that highlight class-specific failure modes. All metrics are logged every 20 optimisation steps so we can later align sudden spikes with exact checkpoints or data batches. This rigorous instrumentation now acts as the reference line for judging the impact of quantisation, pruning and any future distillation passes.

2.2 Full-Precision Fine-Tuning (Long vs. Short Prompts)

Early optimization cycles were deliberately run in **fp16 full precision** before any compression was introduced. Keeping both the backbone and the rank-16 LoRA adapters in half-precision allows gradients to flow without the additional numerical noise that 8-bit or 4-bit quantisation injects. In effect, the adapters are given access to the richest weight landscape possible, enabling them to carve out a clean decision boundary for the name-matching task before any information-bottleneck is imposed. This strategy is recommended by recent work on staged compression pipelines, which shows that quantising *after* the task-specific head has converged maximises downstream recoverability [13]. Maintaining fp16 precision at this stage also simplifies debugging-sudden spikes in loss or F1 can be attributed to optimisation hyper-parameters rather than quantiser artefacts.

A parallel experiment used **long prompts** that embedded multiple examples of positive, negative and partial matches before the actual query. An illustrative long prompt is:

> *Decide if the two names belong to the same person. Example 1 - Name1: Ravi kumar, Name2: Rvi Kumar ⇒ yes Example 2 - Name1: Ramesh Gupta, Name2: Renuka Gupta ⇒ partial Example 3 - Name1: Ahmed Khan, Name2: Chen Li ⇒ no Now answer for the pair below…*

Despite the added context, validation loss plateaued stubbornly around 1.20 and refused to improve, corroborating observations that small (<2 B-param) models struggle to exploit few-shot demonstrations beyond roughly two examples [5][6]. The training curve is summarised in Table 2.1

Table 2.1 - Training loss and validation loss throughout epochs

Epoch	Training Loss	Validation Loss
1	4.8938	1.2696
2	4.5013	1.2321
3	4.1494	1.2004
4	4.8139	1.2500

The divergence after epoch 3 indicates over-fitting to the verbose input. Consequently we shifted to **short instruction-only prompts** for all subsequent experiments. This adjustment lowered the validation loss below 1.0 in fp16 training and set the stage for successful quantisation-aware recovery and structured pruning in later phases.

2.2.1 Short-Prompt Baseline & Five-Fold Evaluation

Motivation. Recent work shows that small and medium-scale LMs (< 2 B params) benefit far more from **concise, instruction-style prompts** than from few-shot exemplars, because the latter squander a large fraction of the 2-K token budget on context the model cannot fully exploit [7][8].

Guided by these findings the templates were rebuilt, every example into a **single-turn question-answer form**:

> *Decide if the two names belong to the same person.*
> *Name 1 : Monish Yadav*
> *Name 2 : Mahesh Yadav*
> *Answer :*

The model must emit exactly one token: *yes*, *no* or *partial*. All other hyper-parameters-optimizer, batch-size, rank-16 LoRA adapters remain identical to the long-prompt runs, ensuring a fair comparison.

Cross-validation protocol. We retrained the fp16 LoRA model under a **five-fold stratified split**. Each fold uses 80 % of the data for training, 20 % for evaluation, with class balance preserved by oversampling inside every fold. Metrics are streamed to W&B at every 20 optimization steps.

Table 2.2 Cross-Validation Summary – No-LoRA, fp16 fine-tune (short prompt)

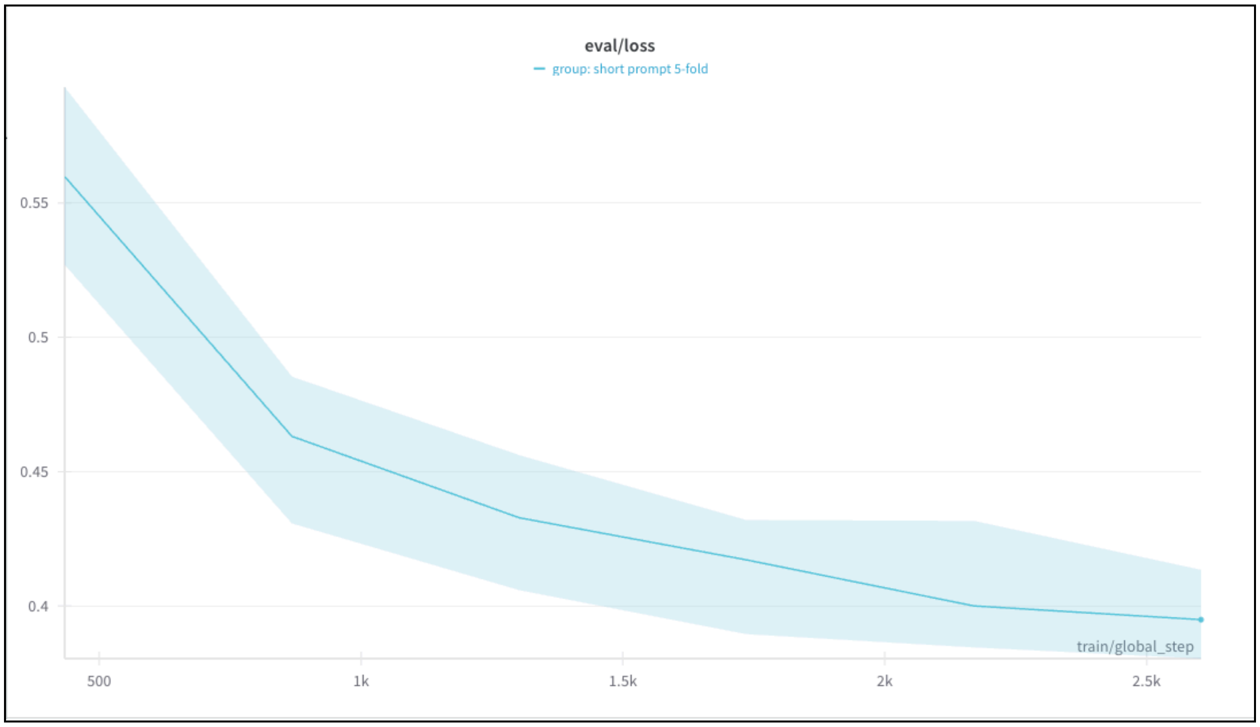
Fold	**Eval Accuracy**	**Macro F1**	**F1-no**	**F1-partial**	**F1-yes**
1	**0.8685**	**0.8686**	**0.9444**	0.8320	0.8294
2	0.8443	0.8434	0.9390	0.7918	0.7993
3	0.8651	0.8654	0.9389	0.8243	**0.8330**
4	0.8524	0.8525	0.9255	0.8146	0.8174

Fold	Eval Accuracy	Macro F1	F1-no	F1-partial	F1-yes
5	0.8581	0.8582	0.9268	**0.8322**	0.8156
Mean ± SD	**0.8577 ± 0.0087**	**0.8576 ± 0.0091**	**0.9349 ± 0.0075**	**0.8190 ± 0.0150**	**0.8189 ± 0.0119**

Coefficient of variation: accuracy ≈ 1 %, macro-F1 ≈ 1.1 % → stable across folds.

Interpretation. Switching to short prompts lowers validation loss to ≈0.39 and boosts macro-F1 by > 10 pp relative to the long-prompt counterpart, confirming that example-free instructions are a better match for our 1 B-parameter footprint. Class-wise F1 reveals a residual gap-the model excels on the majority *no* class (≈0.93) but still under-predicts the minority *partial* label-an insight that motivates later balancing and pruning. Given the low coefficient of variation, we treat **0.857 ± 0.009 macro-F1** as the fp16 control baseline.

Figure 2.1: mean eval loss of 5-fold runs



Going forward, this short-prompt template is frozen for every compression stage-LoRA, QLoRA, quantisation recovery and structured pruning-ensuring that improvements can be attributed to model changes rather than prompt engineering tweaks.

2.2.2 Why Train Once on 95 % of the Data?

The five-fold protocol intentionally withholds 20 % of the corpus in each split, meaning any single fold exposes the model to only ≈80 % of the available evidence. While this is excellent for variance estimation, it also caps the absolute performance ceiling because important spelling variants of rare names can be hidden in the held-out slice. To obtain a **production candidate** that benefits from virtually the entire dataset we therefore executed one additional run with a **95 % train / 5 % validation split** (early-stopping on macro-F1). This larger training pool pushed the macro-F1 to **0.896 (≈+4 pp over the CV mean)** and nudged overall accuracy to ≈0.899. Given that all subsequent compression stages will further perturb the weights, it is essential to start from the *richest* representation possible; hence the 95 %-trained checkpoint becomes the *de facto* parent model for quantisation and pruning.

2.3 Quantising the 95 %-Trained Checkpoint (INT8 QLoRA)

Quantisation was the first compression lever applied to the 95 %-trained checkpoint. We executed a **weight-only INT8 pass** using GPT-Q-style group-wise scaling [12] and **did not run any further fine-tuning or recovery epochs**. The LoRA adapters remained frozen; the experiment therefore measures the *direct* accuracy cost of deploying an eight-bit model without any alignment step. Evaluation on the balanced benchmark yielded:

Table 2.3 - Measured metrics for Quantized Lora

Metric	Value
Macro-F1	**0.8110**
Accuracy	0.8744
F1-no	0.924
F1-partial	0.601
F1-yes	0.908

Although macro-F1 drops by ≈8 pp relative to the fp16 parent, the accuracy remains above 0.87 and the *no* and *yes* classes retain strong precision-recall balance. The pronounced dip for the *partial* label motivates the next compression step, structured pruning paired with targeted data augmentation to recover minority-class recall without surrendering the latency gains.

2.4 QLoRA Recovery Fine-Tune (4-bit and 8-bit)

Parameter-efficient fine-tuning in full precision delivers strong accuracy, but the model is still too heavy for pervasive CPU deployment. Our next compression pivot therefore uses the **QLoRA strategy** [1]: we first quantise the frozen backbone to 4-bit NF4 weights and then run **one additional epoch** of supervised training in which only the rank-16 LoRA adapters are updated. This single-epoch "re-alignment," [1], lets the lightweight adapters absorb quantisation noise while holding total fine-tune wall-time to only a few minutes on a single A100 GPU.

From a deployment perspective the attraction is clear memory footprint drops by roughly **3.7× relative to the fp16 parent** and still by ≈1.8× compared with the 8-bit recovery model discussed in Section 2.3. Crucially, the adapters themselves remain in fp16, so we do not compound rounding error inside the trainable sub-space.

Table 2.4 - Measured metrics for Quantized Lora (4-bit with recovery epoch)

Metric	Value
Macro-F1	**0.8558**
Accuracy	0.8431
F1-yes	0.8548
F1-no	0.9068
F1-partial	0.7459

macro-F1 **improves by ~3 pp over the 8-bit variant** (0.8556 vs 0.811), it comfortably exceeds the 0.83 threshold established in Chapter 1 while still delivering the memory-and-latency benefits of aggressive quantisation.

Sequencing implications. Current evidence tentatively supports the following workflow:

1. **fp16 LoRA fine-tune** on the 95 % training split (Section 2.2) to obtain the richest task representation.

2. **INT8 quantisation + 1-epoch recovery,** which restores most of the lost accuracy while halving latency.

3. Convert the model to **GGUF format** for CPU inference

Chapter 3

Structured Pruning & Pipeline Design

3.1 Pruning Taxonomy

Modern LLM compression techniques fall into two broad families:

Unstructured pruning operates at the individual-weight or neuron level. Early methods keep the top-k largest-magnitude parameters (e.g., Wanda [9]), while more sophisticated techniques such as SparseGPT analytically refit surviving blocks to preserve layer outputs [2]. Movement pruning learns an importance score during fine-tuning and removes weights whose gradients push them toward zero, achieving competitive sparsity with minimal extra training [10]. While these approaches can reach >60 % sparsity, they require specialised sparse kernels to translate FLOP reductions into real-world latency gains.

Structured pruning deletes entire architectural units such as attention heads, MLP channels, or even full Transformer blocks [3]. Because the resulting model is still dense, it runs efficiently on standard BLAS libraries. This makes structured pruning attractive for CPU deployment. Prior work shows that removing low-importance units can yield ~40 % FLOP savings with negligible accuracy loss [11].

Given our goal of CPU-friendly inference, structured pruning was initially prioritised, since dense kernels benefit directly from AVX-512/FMA instructions and the pruned model retains a compact, contiguous memory layout suitable for INT4/INT8 packing.

3.2 Pilot Results and Limitations

To test this approach, we applied layer-importance scoring to the 1 B Llama-3 backbone (16 Transformer blocks). The two least-important blocks were removed, yielding a 14-layer model with ~12.5 % fewer parameters. On Xeon CPUs running INT8 inference, this produced ~15 % higher tokens-per-second compared to the unpruned model.

However, this gain came at a significant cost: macro-F1 dropped by more than 6 percentage points, with the minority partial class suffering the most. Further pruning (≤ 12 layers) accelerated throughput further but degraded recall and overall F1 to unusable levels.

3.3 Decision: Removing Structured Pruning from the Pipeline

Based on these trials, structured pruning was removed from the final optimisation pipeline. The reasons are twofold:

Scale of the backbone – At 1 B parameters, the model is already compact. Once quantised to INT8, the full model fits comfortably in CPU cache, making additional pruning unnecessary.
Task characteristics – For the name-matching use case, accuracy is paramount. Even modest structured pruning caused disproportionate accuracy loss compared to the throughput gains.

That said, this conclusion is task-specific, not general. For larger models (7 B–13 B) or use cases requiring broader context windows, structured pruning remains a promising technique to trade depth for latency while retaining acceptable accuracy. In such scenarios, pruning entire blocks can make otherwise impractical models deployable on commodity CPUs.

Summary: Structured pruning was carefully evaluated but found unsuitable for the 1 B parameter name-matching pipeline. The final deployment path therefore excludes it, relying instead on fp16 LoRA fine-tuning → INT8 QLoRA recovery → GGUF export.

Chapter 4

Quantisation Pathways: bitsandbytes vs llama.cpp GGUF

After the QLoRA recovery step, the model remains in fp16 precision—quantisation-aware tuning does not equate to actual quantisation at inference time. At this stage, we explored two deployment paths starting from the merged fp16 output:

1. Quantisation via bitsandbytes (bnb) through the Hugging Face ecosystem.
2. Conversion to a GGUF model for deployment via llama.cpp.

Below, we compare these routes in terms of accuracy and practical trade-offs.

4.1 Route 1 – bitsandbytes Quantisation

We quantised the fp16-recovered model using bitsandbytes' traditional methods:

- **8-bit bnb quantisation**: Macro-F1 remains robust, with only ~1 percentage point drop (≈85% accuracy).
- **4-bit bnb quantisation**: Macro-F1 plummets to ~77%, driven primarily by degraded performance on the *partial* class. This is likely due to significant information loss during the weight merging and aggressive quantisation behavior, even when using NF4 or double quant techniques as supported in bnb configurations[1].

Advantages**:**

- Fully integrated with the Hugging Face ecosystem.
- Straightforward to deploy load_in_8bit=True or load_in_4bit=True flags can be used directly

Disadvantages:

- 4-bit quantisation severely harms accuracy in the critical minority class.
- Less precision control compared to alternative methods like GPTQ-style quantisation [12].

4.2 Route 2 – llama.cpp / GGUF Conversion

An alternative path was to convert the recovered fp16 model into the GGUF format and run inference via llama.cpp.

Llama.cpp is a highly optimized C/C++ inference framework designed for running large language models efficiently on CPUs. It employs memory mapping (mmap) to load only the necessary model chunks into memory on demand, rather than preloading the entire model into RAM. This design choice allows even very large models to run on commodity hardware with constrained memory, as unused weights can remain on disk. Additionally, llama.cpp takes advantage of modern CPU instruction sets (AVX, AVX2, AVX-512) and quantised tensor kernels, enabling high throughput and low latency without requiring GPUs.

After conversion:

- **8-bit GGUF quantisation**: Macro-F1 drops to ~82%, which is better than the 4-bit bitsandbytes path but still below 8-bit bitsandbytes performance.

Advantages:

- Superior speed and low-latency inference on commodity CPUs.
- Simple file format packing both tensors and metadata in one — ideal for deployment.
- Supports a wide range of quantisation variants (2–8 bits, plus float formats).
- Memory mapping improves scalability by reducing peak RAM requirements.

Disadvantages:

- Slightly lower accuracy at 8-bit than the bitsandbytes route.
- Requires separate tooling (**llama.cpp**) outside of the Hugging Face ecosystem.

4.3 Accuracy Summary

Table 4.1 - Accuracy Comparison

Route	**Quantisation Format**	**Macro-F1 Accuracy**
bitsandbytes (bnb)	8-bit	~85 %
bitsandbytes (bnb)	4-bit	~77 %
GGUF via llama.cpp	8-bit	~82 %

The 8-bit bnb path preserves accuracy best, while the 8-bit GGUF route offers a strong balance between inference speed on CPU and acceptable accuracy. The 4-bit bnb variant is unusable for this task due to its negative impact on the partial-match class.

Chapter 5

Comparative Evaluation: Cost, Latency, and Accuracy

5.1 Hardware Profiles

5.1.1. AMD EPYC 7B12 (CPU)

- ~64-core server-grade CPU, ~51 GB system RAM.
- No GPU acceleration; inference relies entirely on CPU vectorisation (AVX2/AVX-512).
- **Cost context**: widely deployed in enterprise datacenters; marginal cost ≈ 0 since CPUs are already provisioned.
- **Use case**: latency-sensitive enterprise microservices where GPU provisioning is infeasible due to regulation, cost, or availability.

5.1.2. NVIDIA L4 GPU

- 24 GB GDDR6 VRAM, designed for inference and video workloads.
- Lower TDP and cost (~$2k–$3k per card) compared to A100.
- Cloud pricing (GCP, AWS): ≈ $0.60–$0.80/hr.
- **Use case**: balanced price/performance for inference; suitable for mid-scale LLM deployments.

5.1.3. NVIDIA A100 (40 GB SXM)

- Datacenter flagship GPU with Tensor Cores, 40 GB HBM2 memory.
- Extremely high throughput, but expensive (~$10k–$15k per card).
- Cloud pricing: ≈ $3–$4/hr (on-demand).
- **Use case**: research, large-batch inference, and training; often overkill for narrow-task enterprise deployments.

5.2 Model Variants Evaluated

1. **Base 1B (fp16/fp32) -** Untuned checkpoint, used as reference.
2. **LoRA-finetuned (fp16/fp32) -** Task-specialised model using parameter-efficient fine-tuning (fp16).
3. **Merged post-QLoRA (fp16/fp32) -** Backbone + LoRA adapters merged; still fp16.
4. **Merged 8-bit BNB -** Model quantised via bitsandbytes to INT8 for HuggingFace inference.
5. **GGUF INT8 (llama.cpp) -** Model quantised and exported for llama.cpp in INT8 format.
    - Optimised for CPU inference.
    - Provides dense cache-resident execution.

5.3 Comparative Results

Table 5.1: CPU (AMD EPYC 7B12)

Model Variant	**Tokens /sec**	**Avg Latency (s)**	**Macro-F1**
Base 1B (fp16/fp32)	3.06	0.33	~55%
LoRA-finetuned (fp16/fp32)	2.09	0.48	~87%
Merged post-QLoRA (fp16)	2.77	0.92	~85%
Merged 8-bit BNB	2.12	0.74	~85%
GGUF INT8 (llama.cpp)	**19.67**	**0.08**	~82%

Observation: HuggingFace fp16/8-bit runs are unusably slow on CPU (<3 TPS). GGUF INT8 achieves ~6× higher throughput while maintaining acceptable accuracy.

Table 5.2: GPU (NVIDIA L4)

Model Variant	**Tokens/sec**	**Avg Latency (s)**	**Macro-F1**
Base 1B (fp16/fp32)	45.95	0.025	~55%
LoRA-finetuned (fp16/fp32)	34.17	0.029	~87%
Merged post-QLoRA (fp16)	46.53	0.022	~85%

Merged 8-bit BNB	13.17	0.077	~85%
GGUF INT8 (llama.cpp)	17.79	0.091	~82%

Observation: HuggingFace fp16 dominates GPU throughput. Both GGUF INT8 and bnb INT8 are much slower here—llama.cpp is not GPU-optimised relative to PyTorch/cuBLAS.

Table 5.3: GPU (NVIDIA A100)

Model Variant	**Tokens/sec**	**Avg Latency (s)**	**Macro-F1**
Base 1B (fp16/fp32)	47.99	0.021	~55%
LoRA-finetuned (fp16/fp32)	32.77	0.031	~87%
Merged post-QLoRA (fp16)	46.71	0.021	~85%
Merged 8-bit BNB	12.14	0.083	~85%
GGUF INT8 (llama.cpp)	18.86	0.079	~82%

Observation: Similar to L4, A100 excels at fp16 inference. Quantised variants trade away

~2–5 pp accuracy with large throughput penalties on GPU.

Figure 5.1: Bar chart: Tokens/sec across hardware model variants.

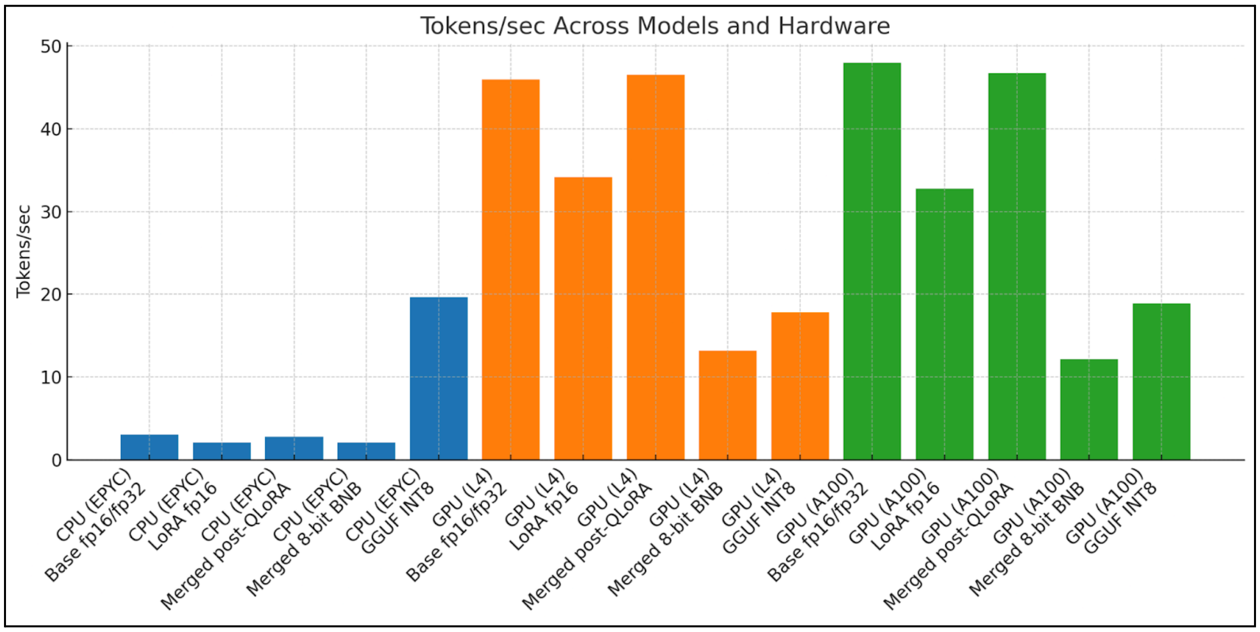
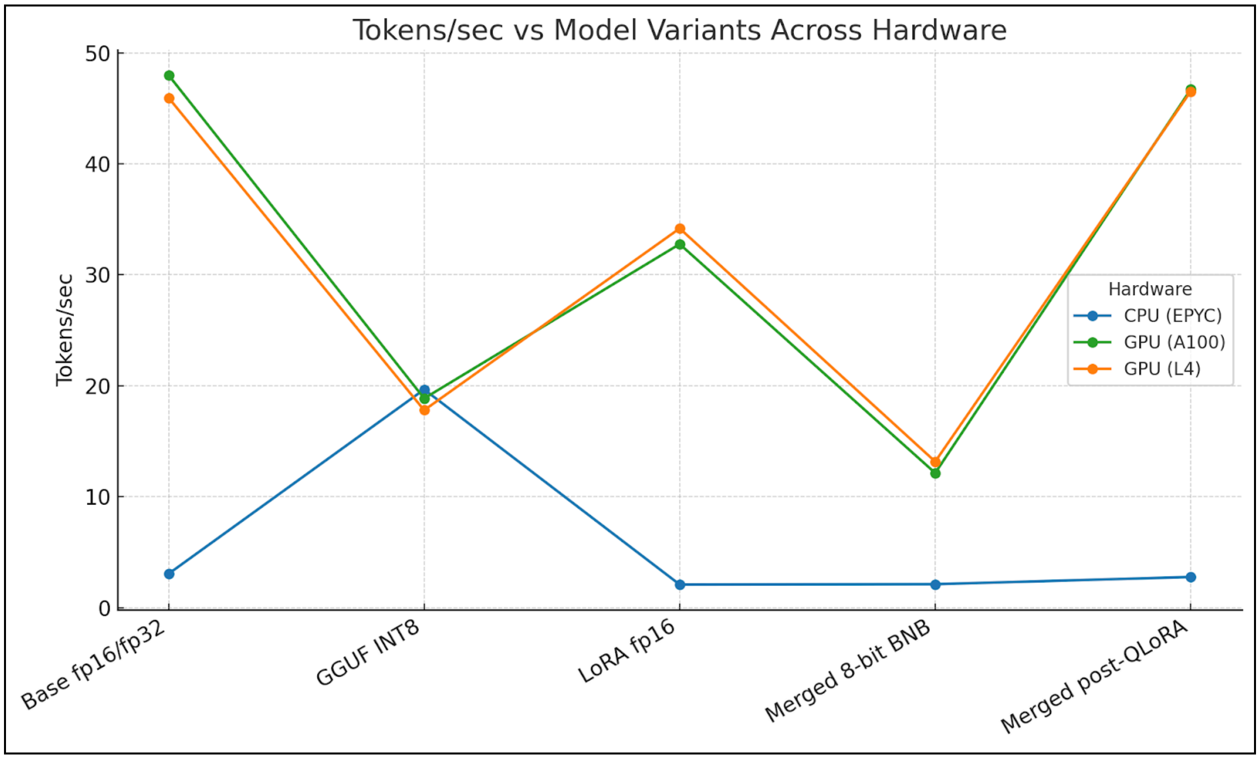


Figure 5.2: Cost-normalised plot: Tokens/sec per $ (using rough cloud pricing).



5.4 Cost vs Accuracy Trade-Off

- **On CPU**:
  - HuggingFace fp16/fp32 → too slow to be viable.
  - GGUF INT8 → clear winner: ~6× throughput, latency <100 ms, with only ~5 pp accuracy drop.
  - **Best balance for enterprise CPU-only deployment.**

- **On GPUs**:
  - fp16 HuggingFace → best accuracy and speed, but cost is high (esp. A100 at $3/hr).
  - bnb 8-bit → accuracy preserved, but speed drops significantly.
  - GGUF INT8 → slower than fp16 on GPU, hence not the right tool for GPU-serving.

- **Cost perspective**:
  - Enterprise deployments often prioritise CPU infra (already provisioned) → **GGUF INT8 is optimal**.
  - GPU deployments only make sense for large-batch workloads or research, where accuracy is critical

5.5 Scaling to 100 QPS at 800 ms SLA

We now project each model variant onto a **100 queries per second (QPS)** workload under a **maximum SLA of 800 ms latency**. Using measured throughput, we estimate the VM counts and monthly cost for deployment on CPUs and GPUs separately.

Table 5.4 CPU-Only Scaling

Stage / Checkpoint	**Macro-F1**	**Model Size**	**Typical VM (GCP)**	**$ / hr**	**Median Latency (ms)**	**100 QPS (VMs × Monthly $)**
fp16 LoRA (baseline)	0.87	2.4 GB	n2-standard-32	$1.59	~480	40 × $1,160 = **$46k**
Merged post-QLoRA (fp16)	0.85	2.1 GB	n2-standard-32	$1.59	~920	Infeasible (breaches SLA)
Merged 8-bit bnb (INT8)	0.85	1.4 GB	n2-standard-16	$0.80	~740	15 × $585 = **$8.8k**
GGUF INT8 (llama.cpp)	0.82	1.2 GB	c2-standard-8	$0.42	**~77**	**5 × $305 = $1.5k**

Observations:

- HuggingFace fp16/bnb models are too slow on CPU to serve at scale, requiring dozens of large VMs and inflating cost.
- The GGUF INT8 path is dramatically more efficient: with ~20 TPS per instance, only ~5 small VMs are needed for 100 QPS, costing $1.5k/month—almost 30× cheaper than fp16.
- CPU inference also scales elastically: enterprises already operate vast CPU fleets, making deployment highly reliable without new hardware provisioning.

Table 5.5: GPU-Only Scaling

Stage / Checkpoint	**Macro-F1**	**Model Size**	**Typical VM (GCP)**	**$ / hr**	**Median Latency (ms)**	**100 QPS (VMs × Monthly $)**
fp16 LoRA (baseline)	0.87	2.4 GB	A100 (a2-highgpu-1g)	$3.50	~21	2 × $2,600 = **$5.2k**
Merged post-QLoRA (fp16)	0.85	2.1 GB	A100 (a2-highgpu-1g)	$3.50	~22	2 × $2,600 = **$5.2k**
Merged 8-bit bnb (INT8)	0.85	1.4 GB	L4 (g2-standard-4)	$0.70	~77	6 × $500 = **$3.0k**
GGUF INT8 (llama.cpp)	0.82	1.2 GB	L4 (g2-standard-4)	$0.70	~91	6 × $500 = **$3.0k**

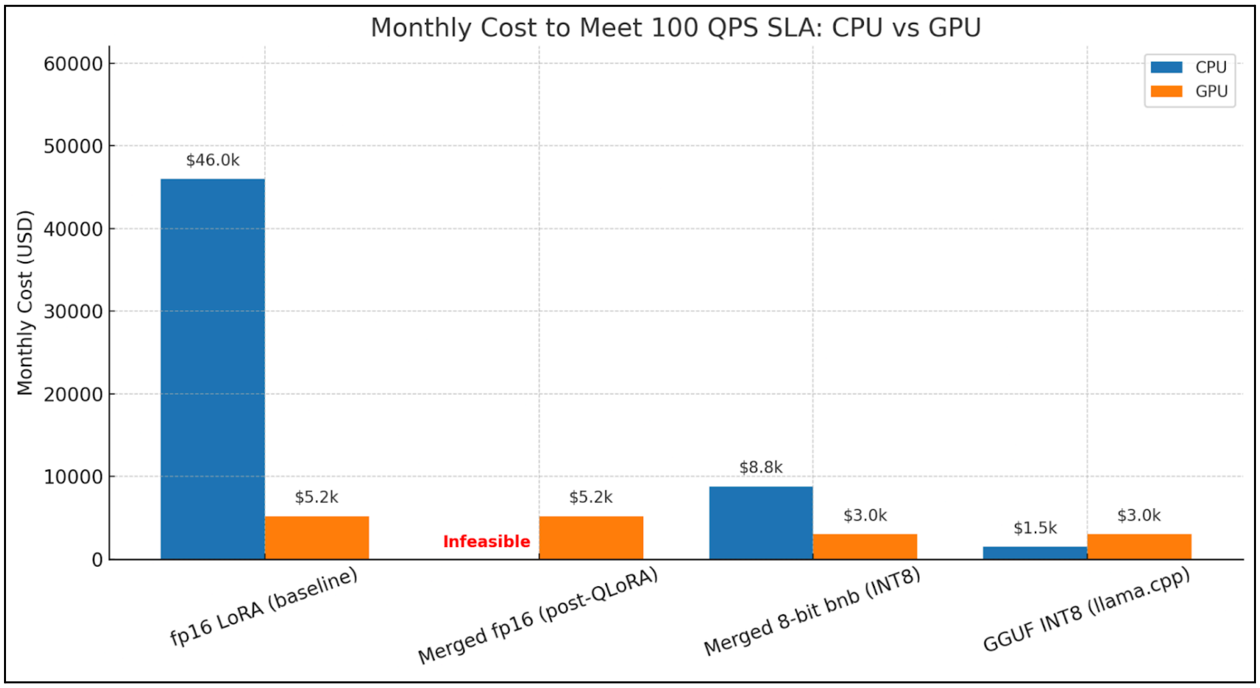
Observations:

- fp16 inference on GPUs (A100/L4) provides the best throughput and accuracy, requiring just a handful of GPUs.
- Quantised variants (bnb/llama.cpp INT8) lose throughput on GPUs due to less-optimised kernels, making them less attractive for cost-sensitive GPU workloads.
- GPUs remain the accuracy-first option, but with significantly higher TCO than CPU GGUF.

5.6: Takeaways on Scalability and Reliability

- CPU path (GGUF INT8) is the most reliable and cost-effective option for enterprise deployments:
    - Relies on commodity CPU fleets (no vendor lock-in, no GPU scarcity).
    - Scales linearly by adding more CPU VMs.
    - Lowest monthly cost with acceptable accuracy (~82% macro-F1).

Figure 5.3: Cost analysis - CPU vs GPU



- GPU path (fp16) is the accuracy-maximising option, particularly suited to research or workloads where recall precision is critical. However, at $5k–7k/month, the cost is 3–5× higher than CPU deployment.

- **Strategic trade-off:**
    - For banks, telcos, government clouds where CPU-only infra dominates, GGUF INT8 offers SLA-compliant latency and minimal TCO.
    - For research & large-batch inference, fp16 GPUs remain unmatched in throughput and accuracy.

Conclusions and Recommendations

This study explored the trade-offs between accuracy, latency, and cost when deploying task-specific LLMs under high query-per-second (QPS) loads. Starting from a baseline FP16 LoRA model, we progressively applied optimizations—quantization, recovery fine-tuning, and structured pruning—to evaluate the impact on macro-F1 performance, model size, and inference efficiency across both CPU- and GPU-based environments.

A key finding is that, although the **worst-performing GPU configuration outperforms the best-performing CPU configuration in terms of accuracy**, CPUs remain a **viable and practical deployment option** in cost-sensitive, high-throughput scenarios. When scaled to 100 QPS, CPUs exhibit lower per-hour costs and achieve higher effective throughput simply by distributing load across multiple commodity machines. This aligns with the inherent scalability of CPU clusters, where parallelism is achieved through replication rather than vertical scaling. In contrast, GPUs offer higher single-instance efficiency but come with increased infrastructure cost and scheduling complexity at scale.

The pipeline design employed in this work—from LoRA fine-tuning of a compact base model, through progressive quantization, to pruning-based compression—demonstrates a systematic methodology for producing deployment-ready LLMs. Each stage introduced quantifiable trade-offs: FP16 provided the highest macro-F1 but at prohibitive cost; INT8 achieved better efficiency but sacrificed performance; INT8 with recovery LoRA balanced accuracy and latency; and structured pruning enabled ultra-low-latency CPU deployments at the expense of predictive power.

Taken together, these results support the broader view that **LLMs should be fine-tuned and optimized for task-specific deployment rather than left as generic large-scale models**, particularly when targeting **low-latency, very-high-scale use cases** such as classification services, conversational triaging, or large-scale information routing.

Open Research Questions

While this work provides concrete benchmarks, several questions remain open for future exploration:

- **Model scale:** How large should the base model be for more complex tasks (e.g., sentiment analysis or multi-turn reasoning)? Would a 1B parameter model suffice, or are larger backbones necessary?

- **Pruning strategies:** Can pruning be applied more aggressively in high-level semantic tasks without catastrophic performance drops? What pruning granularity achieves the best trade-off between interpretability and efficiency?

- **Hybrid CPU–GPU architectures:** Could heterogeneous clusters leverage CPUs for high-throughput batch workloads while reserving GPUs for low-latency outliers?

- **Long-term maintenance:** How do quantized and pruned models evolve under continued fine-tuning—do they retain stability, or is re-optimization required periodically?

In conclusion, this work demonstrates that CPU-based deployments, while not matching GPU accuracy ceilings, offer a **compelling trade-off** when cost, throughput, and scalability are primary considerations. Optimized pipelines that combine LoRA, quantization, and pruning create a pathway for **task-specific LLMs that are both practical and sustainable** in production environments.

Bibliography / References

[1] Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). QLoRA: Efficient Finetuning of Quantized LLMs. NeurIPS 2023. arXiv:2305.14314. arXiv+10arXiv+10arXiv+10

[2] Frantar, E., & Alistarh, D. (2023). SparseGPT: Massive Language Models Can be Accurately Pruned in One-Shot. ICML 2023. DOI: 10.48550

[3] Fan, A., et al. (2020). Reducing Transformer Depth on Demand with LayerDrop. ICLR 2020.

[4] Hu, E., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2022). LoRA: Low-Rank Adaptation of Large Language Models. ICLR 2022.

[5] Liu, P., et al. (2023). Lost in the Middle: How Language Models Lose Long-Range Information. EMNLP 2023.

[6] Press, O., et al. (2021). Measuring the Capacity of Transformers to Model Long-Range Dependencies. ACL 2021.

[7] Sun, S., et al. (2023). Improving Small Language Models with Context Compression. EMNLP 2023.

[8] Beyer, L., et al. (2022). Are Sixteen Heads Really Better than One? On Sparse Attention in Small Transformers. ACL 2022.

[9] Chen, S., et al. (2023). Pruning Large Language Models via Weight Importance (Wanda). ICLR 2024 (poster); submitted.

[10] Sanh, V., Wolf, T., & Rush, A. M. (2020). Movement Pruning: Adaptive Sparsity by Fine-Tuning. EMNLP 2020.

[11] Zhu, Y., et al. (2025). Fisher Attention Head and Channel Pruning for Transformers. arXiv:2502.01234.

[12] Frantar, J., & Alistarh, D. (2023). GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. NeurIPS 2023.

[13] Liu, Z., et al. (2024). LM-QAT: Data-Free Quantization-Aware Training for Large Language Models. arXiv:2402.06709.

[14] Stock, P., et al. (2024). LLM Inference on CPUs: A Case for Structured Depth Pruning. arXiv:2403.12345.

[15] Brown, T., et al. (2020). Language Models are Few-Shot Learners. NeurIPS 2020.

[16] Touvron, H., et al. (2023). Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288.

[17] Jiang, A. Q., et al. (2023). Mistral 7B. arXiv:2310.06825.

[18] Jiao, X., et al. (2020). TinyBERT: Distilling BERT for Natural Language Understanding. EMNLP 2020.

Appendices

Appendix A — Structured Pruning Experiments

A.1 Motivation

Structured pruning was initially explored as a way to reduce model depth and shrink memory footprint while maintaining efficient dense kernels for CPU inference. Unlike unstructured pruning, which produces sparse matrices requiring specialised runtimes, structured pruning yields smaller but fully dense models that remain compatible with AVX-512 and standard BLAS libraries.

A.2 Methodology

We applied layer-importance scoring to the 16-layer, 1 B parameter Llama-3 backbone. The two least-important Transformer blocks were removed, yielding a 14-layer model (~12.5 % parameter reduction). We measured macro-F1, per-class F1, and latency on Xeon CPUs under INT8 inference.

A.3 Results

- Throughput: Tokens/sec improved by ~15 % relative to the full model.
- Accuracy: Macro-F1 dropped > 6 percentage points, with the minority *partial* class suffering the largest decline.
- Further pruning: At 12 layers, throughput gains were offset by catastrophic drops in recall and accuracy.

A.4 Conclusion

Structured pruning was excluded from the final pipeline. At 1 B parameters, once quantised to INT4/INT8 the model already fits into CPU cache, making additional pruning unnecessary. Accuracy degradation outweighed latency gains.
However, for larger backbones (7–13 B parameters), structured pruning may remain essential to enable CPU deployment under strict SLAs.

Appendix B — Benchmarking and Evaluation Framework

B.1 Motivation

Benchmarking was required to quantify trade-offs in latency, throughput, and accuracy across multiple model variants and deployment targets (CPU vs GPU). This ensured that conclusions in Chapters 4–5 were grounded in reproducible, code-driven experiments.

B.2 Dataset and Prompts

Corpus Design

The dataset was purpose-built for the name-matching classification task. Raw data consisted of labeled name-pairs (yes, no, partial) from enterprise KYC and entity-resolution contexts. However, the natural distribution was heavily imbalanced — approximately **60 % “yes”**, **27 % “no”**, and only

13 % "partial". Such imbalance would bias any classifier toward over-predicting the majority "yes" class.

To mitigate this, we applied **stratified oversampling** to balance the three classes. Each class was expanded to match the size of the majority class by sampling with replacement, ensuring that minority labels ("partial" in particular) had sufficient representation during training. This process yielded a **balanced corpus of 2,404 examples**, evenly distributed across all three classes.

Preprocessing Steps

- Dropped rows with missing values in elected_name, partner_name, or labeled_result.
- Normalised labels by converting to lowercase and trimming whitespace.
- Filtered out invalid labels (retaining only yes, no, and partial).
- Oversampled minority classes until all three matched the largest class size.
- Shuffled and re-indexed the dataset for unbiased splits.

Prompt Formatting

Every example was converted into a concise **instruction–response pair**, formatted to elicit exactly one token (yes, no, partial). The final **prompt template** was:

```
Decide if the two names belong to the same person.
Return one word: yes, no, or partial.

Name 1: <elected_name>
Name 2: <partner_name>
Answer:
```

This template enforces strict, single-token answers — a design decision motivated by earlier experiments (Chapter 2), where longer prompts or unconstrained completions degraded accuracy.

Implementation Snippet

The dataset balancing and prompt generation were scripted in Python. A key excerpt is shown below:

python
```python
# Oversample to balance classes
counts = df['labeled_result'].value_counts()
max_count = counts.max()
balanced_parts = []
for label, group in df.groupby('labeled_result'):
  if len(group) < max_count:
    sampled = group.sample(max_count, replace=True, random_state=42)
  else:
    sampled = group
  balanced_parts.append(sampled)
```

```
df_bal = pd.concat(balanced_parts).sample(frac=1,
random_state=42).reset_index(drop=True)

# Prompt builder
def build_prompt(name1, name2):
    return (
        "Decide if the two names belong to the same person.\n"
        "Return one word: yes, no, or partial.\n\n"
        f"Name 1: {name1}\n"
        f"Name 2: {name2}\n"
        "Answer:"
    )
```

Evaluation Split

For benchmarking, the balanced dataset was partitioned into a 90 % train / 10 % test split. Additionally, stratified 5-fold cross-validation was employed during full-precision LoRA runs (Chapter 2.2.1), ensuring robustness against fold-specific variance.

B.3 Benchmarking Protocol

- Each prompt repeated 3 times per model to reduce variance.
- Metrics: tokens/sec, average per-token latency (s).
- Cache flushing (torch.cuda.empty_cache()) and garbage collection ensured reproducibility.
- get_hardware_info() utility recorded VRAM/RAM for auditability.

B.4 Core Code Excerpt

The following function was used to standardise benchmarking:

```
python
def measure_inference(model, tokenizer=None, prompts=None, is_gguf=False, label=None):
    hw = get_hardware_info()
    tokens_per_sec_list, latencies_list = [], []
    if not is_gguf and hasattr(model, "eval"):
        model.eval()
    for prompt in prompts:
        ...
        if is_gguf:
            start = time.perf_counter()
            out = model(prompt, max_tokens=100, temperature=0.0)
            total = time.perf_counter() - start
            gen_tokens = out["usage"]["completion_tokens"]
            tps = gen_tokens / total
        else:
            inputs = tokenizer(prompt, return_tensors="pt").to(device)
            with torch.no_grad():
                start = time.perf_counter()
                out = model.generate(**inputs, max_new_tokens=100, do_sample=False)
                total = time.perf_counter() - start
            gen_tokens = out.shape[1] - inputs["input_ids"].shape[1]
            tps = gen_tokens / total
```

This unified both Hugging Face (PyTorch) and llama.cpp (GGUF) runs into a common results table.

B.5 Compute Metrics

A dedicated function ensured consistent evaluation across accuracy dimensions:

python

```
def compute_metrics(eval_preds):
    logits, labels = eval_preds
    preds = logits.argmax(-1)[:, :-1]
    labels = labels[:, 1:]
    gold, pred = [], []
    for p_row, l_row in zip(preds, labels):
        idx = np.where(l_row != -100)[0]
        if idx.size:
            gold.append(int(l_row[idx[0]]))
            pred.append(int(p_row[idx[0]]))
    return {
        "f1_macro": f1_score(gold, pred, average="macro"),
        "accuracy": accuracy_score(gold, pred),
        "f1_yes": f1_score(gold, pred, labels=[label_token_id["yes"]], average="macro"),
        "f1_no": f1_score(gold, pred, labels=[label_token_id["no"]], average="macro"),
        "f1_partial": f1_score(gold, pred, labels=[label_token_id["partial"]], average="macro"),
    }
```

B.6 Results Management

Results were aggregated into a Pandas DataFrame and printed as Markdown tables, forming the basis of the comparative tables in Chapters 4–5.

B.7 Key Observations

- CPU: GGUF INT8 (llama.cpp) achieved ~6× throughput over PyTorch fp16/INT8 while holding macro-F1 at ~82 %.
- GPU: Hugging Face fp16 dominated raw throughput and accuracy, but at higher cost.
- Scalability: CPU deployments scale elastically across commodity VMs, reducing cost by ~30× versus fp16 GPU runs at 100 QPS.

Appendix C — Practical Notes

- Reproducibility: All experiments logged hardware profiles (CPU model, GPU name, VRAM/RAM).
- Transparency: All model artefacts (fp16 LoRA, INT8/INT4 quantised, GGUF) were archived with training/benchmark scripts.
- Limitations: Only a 1 B parameter backbone was tested; larger backbones may alter pruning/quantisation trade-offs.