



DEMOCRITUS UNIVERSITY OF THRACE
SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS

Research and Development of Energy-Efficient Inference Methods for Generative AI (LLMs) on Low-Power Embedded Systems

UNDERGRADUATE DISSERTATION

Full Name of the author, RN:

Dimitrios Nektarios Vatousis, 220007

Supervisor: Dimitrios Karampatzakis, Associate Professor, Department of
Informatics, Democritus University of Thrace

Kavala, 2026



DEMOCRITUS UNIVERSITY OF THRACE
SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS

Research and Development of Energy-Efficient Inference Methods for Generative AI (LLMs) on Low-Power Embedded Systems

UNDERGRADUATE DISSERTATION

Full Name of the author, RN:

Dimitrios Nektarios Vatousis, 220007

A dissertation submitted in partial fulfilment of the requirements for the undergraduate degree in the Department of Informatics, Democritus University of Thrace.

COMMITTEE OF EXAMINERS

Supervisor: Dimitrios Karampatzakis, Associate Professor, Department of Informatics, DUTH

Member 2: Dr. Petros Amanatidis, Adjunct Professor, Department of Informatics, DUTH

Member 3: Vasileios Tsoukalas, ETEP, Department of Informatics, DUTH

Kavala, 2026

All rights reserved

Copyright © Dimitris Vatousis, 2026

The approval of the undergraduate dissertation by the Department of Informatics, Democritus University

of Thrace, does not necessarily indicate the acceptance of the views of the author.

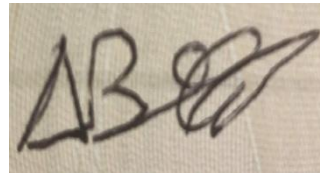
Fulfillment of academic principles and ethics

The present undergraduate dissertation is original and was prepared entirely by the student Dimitris Vatousis of the Undergraduate Program of Studies of the Department of Informatics, under the guidance of the supervisor.

The student certifies that during the preparation and writing of this undergraduate dissertation he/she complied with the provisions of the law and the corresponding internal regulations of the Department, as well as that he/she fully respected the Principles of Academic Ethics and the Code of Ethics, which prohibit falsification of research/experimental results, false reporting, misuse of third-party intellectual property and plagiarism.

The author

Dimitris Vatousis

A handwritten signature in dark ink on a light-colored, textured background. The signature is stylized, starting with a large 'D' and 'V' followed by a series of loops and a long horizontal stroke.

ABSTRACT

This undergraduate thesis focuses on the study, comparison, and development of optimization techniques for the execution of Generative Artificial Intelligence models, specifically Large Language Models (LLMs), on low-resource embedded systems. The work begins with an extensive literature review conducted through the Scopus and IEEE Xplore databases, aiming to analyse state-of-the-art techniques such as model quantization, neural network pruning, and dynamic precision scaling, which contribute to improving energy efficiency during inference.

The experimental procedure is based on the low-power Raspberry Pi 400 platform, combined with an external Atorch UD18 power meter for hardware-level energy measurement. In parallel, a software framework for real-time performance evaluation and monitoring is developed, implemented mainly in Python, using the llama-cpp-python library for inference execution and psutil for system monitoring, while PyTorch and Hugging Face Transformers were used for model preparation.

The primary objective of this thesis is to investigate the feasibility, limitations, and performance trade-offs of executing LLMs in low-power environments, and to propose strategies that enable stable, energy-efficient operation. Experiments with three models (TinyLlama-1.1B, Llama-3.2-1B, Qwen2-0.5B) and four methods (fp16, INT8 Q8_0, 30% unstructured pruning and DPS) showed that the power draw of the Raspberry Pi 400 remains nearly constant (8.1 to 8.5 W), so energy is determined mainly by execution time. INT8 quantization reduced energy per token by about 45% across all models, unstructured pruning without retraining yielded no benefit, and DPS adapted differently to each model (0, 1 and 4 precision switches). Finally, model size affected energy more than any optimization technique. These results provide an experimental foundation for future research on thermally-aware dynamic precision scaling.

SUBJECT AREA: Artificial Intelligence and Embedded Systems

KEYWORDS: Large Language Models, Embedded Systems, Energy Efficient Inference, Quantization, Neural Network Pruning, Dynamic Precision Scaling

CONTENTS

CHAPTER 1: INTRODUCTION	1
1.1 Introduction to Energy-Efficient Execution of Generative AI.....	1
1.2 Motivation and Significance of Energy-Efficient Execution of Generative AI.....	1
1.3 Generative AI and LLMs: Current Trends and Challenges.....	2
1.4 Execution of LLMs on Low-Resource Embedded Systems	2
1.5 Review of Related Literature	3
1.5.1 Energy-Efficient Inference	3
1.5.2 Optimisation Techniques for Embedded Systems.....	3
1.6 Hardware and Software Constraints in Low-Power Environments.....	4
CHAPTER 2: THEORETICAL CONCEPTS AND OPTIMISATION METHODS	7
2.1 Fundamental Concepts of Machine Learning and Neural Networks.....	7
2.2 Large Language Models (LLMs)	7
2.2.1 Transformer Architecture.....	7
2.2.2 Training and Inference in LLMs	8
2.3 Optimisation Techniques for Energy-Efficient Inference.....	8
2.3.1 Model Quantization.....	9
2.3.2 Neural Network Pruning.....	9
2.3.3 Dynamic Precision Scaling	10
2.4 Thermally-Aware and Energy-Aware Inference	11
CHAPTER 3: HARDWARE SYSTEM DESCRIPTION.....	14
3.1 System Hardware	14
3.1.1 Hardware Specifications.....	14
3.2 Computational and Energy Constraints of the System	14
3.2.1 Computational Constraints	14
3.2.2 Energy Constraints.....	15
3.3 Energy Consumption Measurement System	16
3.3.1 Description of the Atorch UD18	16
3.4 System Wiring and Diagram	17
CHAPTER 4: SOFTWARE DESCRIPTION	20
4.1 Choice of Programming Languages	20
4.1.1 Python	20
4.1.2 YAML	20
4.2 Libraries and Development Environment	20
4.2.1 Microsoft Visual Studio Code	21
4.2.2 Core Machine Learning and Inference Libraries	21
4.2.3 Monitoring and Measurement Libraries.....	21
4.2.4 Supporting Libraries	21

4.3 Code Description	22
4.3.1 run_method.py Level: High-Level	22
4.3.2 Model Preparation (Windows PC) Level: Low-Level	23
4.3.3 dps_inference (logic within run_method.py) Level: High-Level.....	23
4.3.4 performance_monitor.py Level: Low-Level	23
4.3.5 energy_logger.py Level: Low-Level	24
4.3.6 metrics.py Level: High-Level.....	24
4.3.7 graphs.py Level: High-Level.....	25
4.3.8 experiment_runner.py & config.yaml Level: High-Level.....	25
4.4 Diagrams and Modelling (UML)	26
4.4.1 Use Case Diagram	27
4.4.2 Component Diagram	28
4.4.3 Class Diagram.....	28
4.5 Code Implementation Examples & Application Description.....	29
4.5.1 DPS Switching Logic (run_method.py)	29
4.5.2 Reading the UD18 Sensor (energy_logger.py)	31
4.5.3 Parallel Metric Collection (metrics.py)	32
4.5.4 Experiment Orchestration (experiment_runner.py)	35
4.5.5 Example Terminal Output	36
 CHAPTER 5: EXPERIMENTAL EVALUATION AND RESULTS	37
5.1 System Operation Check	37
5.1.1 Verification of the UD18 Sensor Connection	37
5.1.2 Inference Verification	37
5.1.3 Verification of Experiment Execution	39
5.1.4 Verification of the DPS Mechanism	39
5.2 Summary of Experimental Results	40
5.3 Analysis per Method	40
5.3.1 fp16 Method	40
5.3.2 INT8 (Q8_0) Method.....	43
5.3.3 Pruned Method.....	45
5.3.4 DPS Method	47
5.4 Comparative Analysis of Methods	50
5.5 Effect of Model Size	51
5.6 Final Conclusions	51
 CHAPTER 6: CONCLUSIONS AND FUTURE WORK	52
6.1 Overall Conclusions of the Thesis	52
6.2 Problems Encountered	52
6.2.1 Redesign of the DPS Logic	52
6.2.2 Decoding the UD18 Protocol	52
6.2.3 Memory Management on the Raspberry Pi	53
6.3 Critical Evaluation.....	53
6.3.1 Strengths of the Implementation.....	53
6.3.2 Limitations and Weaknesses	53
6.4 Future Work and Extensions.....	54

6.4.1 Application to Larger LLMs.....	54
6.4.2 DPS on Faster Hardware.....	54
6.4.3 Research on Thermally-Aware DPS AI Inference.....	54
REFERENCES.....	56
APPENDIX 1. DEVELOPED CODE.....	58

LIST OF FIGURES

Figure 1: Atorch UD18	16
Figure 2: System Wiring & Diagram	17
Figure 3: System.....	18
Figure 4: Use Case Diagram	27
Figure 5: Component Diagram.....	28
Figure 6: Class Diagram.....	28
Figure 7: Verification of Experiment Execution.....	39
Figure 8: tinyllama fp16 Execution Time, Average Power, Total Energy & Energy Efficiency.	41
Figure 9: Average CPU & RAM Usage tinyllama fp16.	41
Figure 10: llama3.2 fp16 Execution Time, Average Power, Total Energy & Energy Efficiency.	42
Figure 11: Average CPU & RAM Usage llama3.2 fp16.....	42
Figure 12: qwen2 fp16 Execution Time, Average Power, Total Energy & Energy Efficiency.....	42
Figure 13: Average CPU & RAM Usage qwen2 fp16.....	43
Figure 14: tinyllama int8 Execution Time, Average Power, Total Energy & Energy Efficiency.	43
Figure 15: Average CPU & RAM Usage tinyllama int8.	43
Figure 16: llama3.2 int8 Execution Time, Average Power, Total Energy & Energy Efficiency.	44
Figure 17: Average CPU & RAM Usage llama3.2 int8.....	44
Figure 18: qwen2 int8 Execution Time, Average Power, Total Energy & Energy Efficiency.....	44
Figure 19: Average CPU & RAM Usage qwen2 int8.....	45
Figure 20: tinyllama pruned Execution Time, Average Power, Total Energy & Energy Efficiency.	45
Figure 21: Average CPU & RAM Usage tinyllama pruned.....	45
Figure 22: llama3.2 pruned Execution Time, Average Power, Total Energy & Energy Efficiency.	46
Figure 23: Average CPU & RAM Usage llama3.2 pruned.....	46
Figure 24: qwen2 pruned Execution Time, Average Power, Total Energy & Energy Efficiency.	46
Figure 25: Average CPU & RAM Usage qwen2 pruned.	47
Figure 26: tinyllama dps Execution Time, Average Power, Total Energy & Energy Efficiency.	48
Figure 27: Average CPU & RAM Usage tinyllama dps.	48
Figure 28: llama3.2 dps Execution Time, Average Power, Total Energy & Energy Efficiency.	49
Figure 29: Average CPU & RAM Usage llama3.2 dps.	49
Figure 30: qwen2 dps Execution Time, Average Power, Total Energy & Energy Efficiency.	49
Figure 31: Average CPU & RAM Usage qwen2 dps.....	50
Figure 32: Comparison of Methods Across All Models	50

LIST OF TABLES

Table 1: Hardware Specifications	14
Table 2: Representative responses per model and method. The fp16, int8 and dps responses are generally coherent. The pruned method shows repetitions in Llama 3.2 and Qwen2.	37
Table 3: Timeline of DPS execution per run for TinyLlama. Blue = fp16, yellow = Q8_0. The switch is decided based on the s/token of the previous run.	39
Table 4: Summary of experimental results, average of 5 runs per model/method combination...	40
Table 5: DPS behaviour per model. Thresholds: slow=0.80 s/token, fast=0.65 s/token.	47
Table 6: Relative change in J/token versus the fp16 baseline.....	50

GLOSSARY OF TERMS

English term	Greek term
Generative Artificial Intelligence	Γενετική Τεχνητή Νοημοσύνη
Large Language Model	Μεγάλο Γλωσσικό Μοντέλο
Inference	Εκτέλεση / Εξαγωγή Πρόβλεψης
Quantization	Κβαντισμός
Pruning	Αποκοπή Νευρωνικών Δικτύων
Dynamic Precision Scaling	Δυναμική Κλιμάκωση Ακρίβειας
Embedded System	Ενσωματωμένο Σύστημα
Energy-Efficient	Ενεργειακά Αποδοτικό
Thermal Throttling	Θερμικός Περιορισμός
Memory Footprint	Αποτύπωμα Μνήμης
Knowledge Distillation	Απόσταξη Γνώσης
Foundation Model	Θεμελιώδες Μοντέλο
Fine-tuning	Λεπτή Ρύθμιση
Pre-training	Προ-εκπαίδευση
Sparsity	Αραιότητα
Activation Function	Συνάρτηση Ενεργοποίησης
Backpropagation	Οπισθόδρομη Μετάδοση Σφάλματος
Overfitting	Υπεραρμογή
Attention Mechanism	Μηχανισμός Προσοχής
Decoder	Αποκωδικοποιητής
Encoder	Κωδικοποιητής
Latency	Καθυστέρηση
Throughput	Απόδοση / Ρυθμαπόδοση
Benchmark	Δοκιμή Αξιολόγησης
Layer	Επίπεδο (Νευρωνικού Δικτύου)
Batch Size	Μέγεθος Δέσμης
Learning Rate	Ρυθμός Εκμάθησης
Token	Λεκτική Μονάδα
Mixed Precision	Μικτή Ακρίβεια
Weight	Βάρος (Παράμετρος Μοντέλου)

ABBREVIATIONS – ACRONYMS

ANN	Artificial Neural Network
API	Application Programming Interface
ARM	Advanced RISC Machine
AWQ	Activation-Aware Weight Quantization
BERT	Bidirectional Encoder Representations from Transformers
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DNN	Deep Neural Network
DPS	Dynamic Precision Scaling
DUTH	Democritus University of Thrace
DVFS	Dynamic Voltage and Frequency Scaling
FFN	Feed-Forward Network
FP16	16-bit Floating Point
FP32	32-bit Floating Point
FP8	8-bit Floating Point
GPT	Generative Pre-trained Transformer
GPTQ	Generative Pre-trained Transformer Quantization
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
INT4	4-bit Integer
INT8	8-bit Integer
IoT	Internet of Things
KV-cache	Key-Value Cache
LLaMA	Large Language Model Meta AI
LLM	Large Language Model
LoRA	Low-Rank Adaptation
LSTM	Long Short-Term Memory
MDP	Markov Decision Process
ML	Machine Learning
MLP	Multi-Layer Perceptron
NLP	Natural Language Processing
PMPD	Progressive Mixed-Precision Decoding

PTQ	Post-Training Quantization
QAT	Quantization-Aware Training
RAM	Random Access Memory
RLHF	Reinforcement Learning from Human Feedback
RNN	Recurrent Neural Network
RoPE	Rotary Position Embedding
SGD	Stochastic Gradient Descent
SLO	Service Level Objective
TAPAS	Thermal- and Power-Aware Scheduling
TAWS	Thermally-Aware Workload Scheduling
TPU	Tensor Processing Unit
TTFT	Time to First Token
UML	Unified Modeling Language
vLLM	Virtual Large Language Model serving framework
YAML	YAML Ain't Markup Language

Chapter 1: Introduction

1.1 Introduction to Energy-Efficient Execution of Generative AI

In recent years there has been rapid progress in the field of artificial intelligence, with Generative Artificial Intelligence models, and in particular Large Language Models (LLMs), at its centre. These models use an architecture based on the Transformer and have demonstrated remarkable capability in natural language processing tasks such as text generation, translation and code generation, making them fundamental tools for modern intelligent applications.

LLMs have demonstrated impressive technical capabilities, at the cost, of course, of increased demands on computational resources, memory and energy consumption. The large number of parameters, the self-attention mechanism and the need to process long data sequences make the execution (inference) of these models particularly energy-intensive, even on modern high-performance computing platforms. We therefore observe that energy efficiency is a critical factor for the sustainable and scalable use of Generative AI.

The problem becomes even more evident when considering the execution of LLMs on Low-Power Embedded Systems, such as single-board computers, devices at the edge of the network and microcontrollers. These systems are characterised by limited computational power, little available memory, strict energy budgets and often tighter thermal constraints. Nevertheless, the growing need for local data processing, reduced latency and privacy protection makes running Generative AI directly at the edge highly desirable or even necessary.

In this context, the energy-efficient inference of LLMs is an active and particularly important research field. The development and application of techniques that reduce energy consumption, without significant degradation of model accuracy and functionality, is decisive for the practical use of Generative AI in low-power environments.

This thesis is situated within this research context, investigating methods and strategies that enable the efficient and thermally aware execution of LLMs on low-resource embedded systems.

1.2 Motivation and Significance of Energy-Efficient Execution of Generative AI

The growing adoption of LLMs has radically transformed the way modern intelligent applications are designed and implemented. Initially, these models were used mainly on Cloud Computing infrastructure, where their heavy demands could be absorbed thanks to the wide availability of computational resources. However, recent trends have increased the need to move processing to the edge of the network, creating new challenges and thus making it necessary to study their execution in low-resource environments [1].

One of the most significant obstacles to deploying Generative AI on embedded systems is energy consumption and, consequently, thermal constraints. Running computationally demanding models on battery-powered devices, or on devices with limited heat dissipation capability, can lead to thermal throttling, reduced performance or even system instability. At the same time, increased energy consumption raises the operating cost of these systems, especially in large-scale applications.

Although there is an extensive body of literature on optimising the use of LLMs on low-resource devices, there is a clear lack of sources on energy- and thermally-efficient execution methods. This thesis aims to fill this gap, as such methods contribute significantly to extending Generative AI to edge environments, increasing the autonomy, reliability and sustainability of future low-power devices.

1.3 Generative AI and LLMs: Current Trends and Challenges

Today's Large Language Models are based on Transformer architectures with hundreds of millions to hundreds of billions of parameters, which allows them to learn general language representations and to adapt to a multitude of different tasks. Models such as these lie at the core of the development of Generative AI, resulting in their rapid growth in both complexity and scale. The trend in the use of LLMs has led to the creation of foundation models, which function as general-purpose natural language processing engines.[2]

As scale has increased, so too has the range of ways in which LLMs are used. Beyond cloud-based service applications, there is now strong interest in local execution (on-device inference) and edge applications, owing to the advantages associated with them. This shift aims at reduced latency, protection of data privacy and improved system reliability. These trends establish LLMs as a key component of smart devices, IoT (Internet of Things) systems and autonomous systems, significantly broadening their field of application.

On the other hand, these developments are accompanied by particular technical challenges. Inference in LLMs is characterised by a high computational load, increased memory requirements and heavy energy consumption, mainly due to matrix multiplication operations and the self-attention mechanism. In low-power environments, such as those addressed by this research, these increased demands often lead to reduced performance, increased thermal stress and limited sustainability of continuous operation. The gap between the ever-increasing complexity of LLMs and the limited capabilities of embedded systems makes the development of new optimisation approaches necessary.

Studies on current trends in Generative AI argue [3] that simply deploying existing models on resource-constrained platforms is not feasible without significant compromises. The study of techniques that reduce the energy and thermal footprint of LLMs is therefore a critical factor for their sustainable integration into future embedded and edge systems.

1.4 Execution of LLMs on Low-Resource Embedded Systems

Running Large Language Models on low-resource embedded systems differs considerably from running them in cloud environments or on high-performance infrastructure. By embedded systems we mean platforms such as single-board computers, a category that includes the Raspberry Pi 400, edge devices based on ARM architectures, and microcontrollers, which are characterised by limited computational power, little available memory and strict energy constraints[3].

Unlike data centres, where LLM execution is accelerated by hardware accelerators such as GPUs and TPUs and supported by adequate cooling systems and a high energy budget, embedded systems usually rely on general-purpose CPUs with a limited number of cores and a small thermal operating margin[4]. The absence of specialised acceleration hardware makes highly complex operations, such as matrix multiplications and the self-attention mechanism, particularly demanding.

A further significant constraint is the memory footprint of LLMs, since storing a model's parameters and the intermediate activations during inference can exceed the RAM available on low-power systems. The problem is aggravated in real-time applications, because low response latency and stable performance are required. In such environments, increased energy consumption often leads to thermal stress and to the activation of thermal throttling mechanisms, which lower the operating frequency of the processor and adversely affect overall performance.

From the above constraints we conclude that simply moving an LLM from a cloud environment to an embedded platform is not feasible without significant modifications. Optimisation techniques tailored to the characteristics of each hardware platform are required, aiming to reduce energy consumption and memory footprint while maintaining an acceptable level of accuracy. The systematic study of LLM behaviour on low-power platforms is therefore a critical step towards their practical use in edge Artificial Intelligence applications.

1.5 Review of Related Literature

This section presents a systematic review of the existing literature related to the main axes of this thesis. Specifically, two thematic areas are examined: energy efficiency during the execution of language processing models, and the optimisation techniques that have been proposed for their execution on resource-constrained embedded systems. The analysis of existing research highlights both the achievements of current work and the open issues that have not yet been addressed, which form the motivation for the present study.

1.5.1 Energy-Efficient Inference

By energy-efficient inference we refer to the optimisation of the prediction process of a machine learning model with the aim of reducing energy consumption, while maintaining acceptable levels of both performance and accuracy. Unlike simply accelerating execution, energy efficiency is assessed through metrics such as energy per execution (joules per inference), energy per generated token, and performance per watt.

The energy analysis of LLMs has emerged as a critical research question in the international literature, particularly in the context of moving them from high-performance environments to edge systems. The work of Islam et al. [5] examines the energy footprint of small language models on edge platforms, noting that inference can exhibit significant fluctuations in power consumption, even for models of reduced size. The results show that reducing the number of parameters does not imply a linear reduction in energy consumption, which highlights the complexity of the relationship between architecture and efficiency. At the same time, the study by Martínez et al. [6] focuses on the analysis of quantised inference on low-power processors, highlighting the significant effect of numerical representation precision on overall energy consumption.

The results of that research show that the use of reduced precision (e.g. INT8) can lead to a significant improvement in energy efficiency without notable degradation of the results, especially in transformer-based architectures.

Overall, the existing literature indicates that the energy cost of LLMs is not due simply to model size, but is also affected by other factors, specifically the type of hardware and the execution strategy or technique. However, most studies focus either on smaller models or on specific optimisation techniques, rather than on a holistic approach that combines energy and thermal analysis in real embedded environments. This gap makes further investigation of energy-efficient LLM execution strategies on low-power systems necessary.

1.5.2 Optimisation Techniques for Embedded Systems

Running LLMs on low-resource embedded systems requires the application of specialised optimisation techniques that aim to reduce energy consumption, memory footprint and execution time while maintaining acceptable model performance. The main categories of techniques proposed in the literature include quantization, neural network pruning, knowledge distillation and dynamic precision adaptation strategies[7][8].

The first technique, quantisation, is now considered one of the most widespread optimisation techniques for low-resource systems. In quantisation, the numerical precision of weights and activations is reduced (for example from FP16 to INT4), thereby achieving a significant reduction in memory footprint and in the computational energy required. The study by Martínez et al. [6] shows that quantised execution of transformer-based models on low-power processors can lead to a noticeable improvement in energy efficiency, especially when suitable vectorised implementations are used.

The next technique is pruning, which aims to reduce the number of parameters of a model by removing non-critical connections or entire structural components. These approaches may be unstructured (unstructured sparsity) or structured (structured pruning), with the latter usually being more hardware-friendly.

Although pruning can reduce model size and theoretical complexity, the practical energy improvement depends on the extent to which the hardware and software can effectively exploit the resulting sparsity.

Finally, dynamic precision scaling works by adapting the numerical precision during execution, based on system load or thermal constraints. This approach directly links energy efficiency to the thermal behaviour of the platform and is a particularly promising direction for systems with a strict energy budget.

Overall, optimisation techniques for embedded systems do not operate in isolation but are often combined in order to achieve an optimal balance between accuracy, performance and energy consumption. The choice of the appropriate strategy depends both on the characteristics of the model and on the constraints of the particular hardware environment.

1.6 Hardware and Software Constraints in Low-Power Environments

The execution of Large Language Models on low-power systems is directly affected by a set of constraints at both the hardware and the software level. These constraints determine the limits of the system's performance, energy consumption and thermal behaviour, and are key factors in the design of energy-efficient inference methods.

Hardware constraints arise because low-power platforms usually rely on general-purpose, low-consumption processors with a limited operating frequency and a small number of cores.

The lack of specialised accelerators, such as GPUs or TPUs, makes the self-attention mechanism and matrix multiplication operations particularly demanding in both time and energy. A further constraint is the lack of sufficient RAM, which results in strict limits on model size and on the intermediate data structures used during inference.

Software constraints, on the other hand, mainly concern issues such as the loss of efficiency caused by default high-precision floating-point arithmetic (e.g. FP16), the overhead of dynamic programming languages and the limitations of backend implementations. In addition, executing models through general-purpose machine learning libraries in CPU environments often fails to exploit the full capabilities of the hardware. Furthermore, on resource-constrained systems, memory management and process scheduling by the operating system may introduce variability in performance.

Energy constraints are also worth mentioning, as they are one of the main problems faced by low-power systems. In systems that operate on a low energy budget or are battery-powered, the momentary increase in consumption while running an LLM can reduce autonomy or trigger power management mechanisms.

Moreover, limited heat dissipation capability means that prolonged execution of computationally demanding tasks may cause thermal throttling, dynamically lowering the operating frequency of the processor and affecting the stability of performance.

Understanding these constraints is essential for developing suitable optimisation techniques, since creating energy-efficient LLM execution methods requires an approach that takes into account the relationship between model architecture, hardware and software.

Chapter 2: Theoretical Concepts and Optimisation Methods

2.1 Fundamental Concepts of Machine Learning and Neural Networks

Machine Learning (ML) is a branch of Artificial Intelligence that enables computing systems to acquire knowledge from data without being explicitly programmed for each task. Artificial Neural Networks (ANNs) are the core computational structure of modern ML systems, inspired by the biological architecture of the human brain.

A neural network consists of layers of artificial neurons: the input layer, one or more hidden layers and the output layer. Each neuron receives a weighted sum of inputs, applies an activation function and passes the result on to the next layer. Deep Neural Networks (DNNs) extend this architecture with multiple hidden layers, allowing the representation of extremely complex non-linear functions[9].

A neural network is trained using the backpropagation algorithm, which computes the partial derivatives of the loss function with respect to each trainable parameter and then updates the weights through gradient-based algorithms such as Stochastic Gradient Descent (SGD) or Adam. Key concepts that shape training include the learning rate, the batch size, the number of epochs and regularization to counter overfitting.

Convolutional Neural Networks (CNNs) introduced hierarchical feature extraction for imaging applications, while Recurrent Neural Networks (RNNs) and their LSTM/GRU variants were widely used for sequential data such as natural language.

However, the limitations of RNNs with respect to parallel processing and long-range data dependencies led to the development of the Transformer architecture, which is presented in the next section.

2.2 Large Language Models (LLMs)

Large Language Models (LLMs) are a category of large deep learning neural networks, trained on vast text collections in order to understand and generate natural language[9]. Architecturally, they are based on the Transformer model and number from hundreds of millions to billions of parameters. Milestones such as OpenAI's GPT (Generative Pre-trained Transformer), Meta's LLaMA and Google's BERT have set new performance standards in Natural Language Processing (NLP) tasks.

The exponential growth in the size of these models simultaneously raises the demands for computational resources and energy, especially during the inference phase.

2.2.1 Transformer Architecture

The Transformer architecture was introduced in 2017 by Vaswani et al. in the paper “Attention is All You Need”[10] and is the fundamental structure of modern LLMs. It replaced recurrent networks in sequence modelling by introducing a new mechanism that processes entire sequences in parallel.

The basic Transformer architecture consists of an encoder and a decoder. Each encoder layer comprises: (a) a multi-head self-attention mechanism, (b) a feed-forward neural network (FFN/MLP), (c) residual connections and (d) layer normalization.

Modern generative LLMs (e.g. GPT-4, LLaMA) mainly use a decoder-only architecture. The self-attention mechanism is the heart of the Transformer[11]. For each token in the input sequence,

it computes a Query (Q), Key (K) and Value (V) vector through trainable projection matrices. Attention is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}(QK^T/\sqrt{dk}) \cdot V$$

where dk is the dimension of the Key vectors. The multi-head attention mechanism performs multiple self-attention operations (“heads”) in parallel, allowing the model to attend to different relationships between tokens simultaneously.

Positional encoding is added to the token representations to encode position information, since the self-attention mechanism on its own is invariant to order. Modern LLMs use techniques such as Rotary Position Embedding (RoPE) or Absolute Position Embedding [12].

The computational complexity of self-attention scales quadratically with the length of the input sequence ($O(n^2)$) [11], which is a major obstacle to efficient inference over long text contexts. This characteristic leads to increased computational demands, significant memory consumption and increased thermal load.

2.2.2 Training and Inference in LLMs

Training an LLM is usually divided into two phases: pre-training and fine-tuning. During pre-training, the model is optimised to predict the next token over a huge text corpus (self-supervised learning), acquiring general linguistic knowledge. This phase requires enormous computational resources, typically thousands of GPUs/TPUs for weeks or months, with energy consumption on the order of GWh.

Fine-tuning adapts the pre-trained model to specialised tasks or domain-specific data. Techniques such as RLHF (Reinforcement Learning from Human Feedback) are used to align the model with human preferences, while LoRA (Low-Rank Adaptation) dramatically reduces the number of trainable parameters during the fine-tuning phase.

Inference refers to the phase in which an already trained model produces a response. In autoregressive LLMs, inference follows two phases:

- Prefill (pre-processing phase): Processes the entire input prompt in parallel. In this phase the KV-caches (Key-Value caches) are computed and stored for each input token. Computationally, it is intensive (compute-bound) due to massive matrix operations.
- Decode (decoding phase): Generates one token at a time autoregressively, using the stored KV-caches. It is characterised by a strong dependence on memory bandwidth (memory-bound), which significantly affects energy consumption.

In practice, the decode phase dominates the total energy consumption of an inference request. The reason is that decode runs once for every generated token, whereas prefill runs once per request, so in tasks with a short prompt and a long response, decode accounts for the great majority of time and energy (see also Section 2.4). This makes finding efficient optimisation techniques for the decode phase critically important.

2.3 Optimisation Techniques for Energy-Efficient Inference

As the deployment of LLMs becomes more widespread, the search for techniques that reduce the computational and energy cost of inference has attracted intense research interest. The main categories of optimisation include model quantization, pruning, knowledge distillation, dynamic precision scaling and architectural optimisations. Quantization, pruning and dynamic precision scaling, which are also examined experimentally in this thesis, are analysed in detail in the following subsections [13].

2.3.1 Model Quantization

Quantization is one of the most widely used inference optimisation techniques. It is based on reducing the numerical precision with which a model's parameters and activations are represented, usually from 32-bit or 16-bit floating-point to 8-bit (INT8) or 4-bit (INT4) integers, with the aim of reducing memory, accelerating computation and saving energy.

There are two main categories of quantization:

- Post-Training Quantization (PTQ)
- Quantization-Aware Training (QAT).

PTQ is applied to already trained models without retraining, converting FP32 weights to lower-precision formats. Owing to its simplicity and efficiency, it is the dominant method in current LLM deployments. QAT incorporates quantization errors into the training process, achieving higher accuracy at a lower bitwidth, but requires additional computational cost.

Important PTQ techniques for LLMs include GPTQ, AWQ (Activation-Aware Weight Quantization) and SmoothQuant. GPTQ uses approximate second-order optimisations to minimise per-layer reconstruction errors, while AWQ takes the distribution of activations into account in order to identify and protect critical weights[14].

Quantization can be applied to different components of the model: to the weights only (weight-only quantization), to both weights and activations (W+A quantization), or also to the KV-cache. Weight-only quantization is the most widespread approach on edge and mobile devices, where memory constraints are critical.

An extensive empirical study evaluating 28 quantised LLMs of the Ollama framework on a Raspberry Pi 4 (4GB RAM) showed that PTQ with 4-bit weights (q4_K_M) achieves an optimal balance between energy consumption, latency and accuracy in resource-constrained environments[15]. The results showed that models with more parameters, once quantised, can outperform smaller non-quantised models in terms of the energy–accuracy balance.

An advanced form of quantization is mixed-precision quantization, in which different layers or components of the model use a different bitwidth depending on their sensitivity to precision reduction. This approach achieves a better balance between accuracy and efficiency than uniform quantization.

Of particular importance is the recent adoption of FP8 (8-bit floating-point) formats by modern accelerators, such as NVIDIA H100 GPUs. The FP8 format offers twice the throughput of FP16, with memory consumption reduced by up to 50%, while maintaining satisfactory accuracy through stabilisation and dynamic scaling techniques[16].

2.3.2 Neural Network Pruning

Neural network pruning is a classic model compression technique that aims to eliminate redundant or insignificant parameters, thereby reducing size and computational cost without significant loss of performance. The number of publications on pruning exceeded 3,000 for the period 2020–2024[17], reflecting the intense research interest.

Pruning techniques are classified into three main types:

- **Unstructured Pruning:** Removes individual weights anywhere in the network, achieving high sparsity ratios with minimal loss of accuracy. However, without specialised hardware support, the practical inference speed-up remains limited.
- **Structured Pruning:** Removes entire filters, channels, attention heads or layers, producing dense models that can be accelerated directly. The LLM-Pruner method, for example, identifies clusters of neurons based on internal dependencies and removes those of low importance to the final output.
- **Semi-Structured Pruning (Semi-Structured / N:M Sparsity):** A compromise between the two previous categories. It enforces sparsity within groups of consecutive weights (e.g. 2:4 sparsity: 2 zeros per 4 weights), achieving a practical speed-up on hardware that supports this pattern, such as NVIDIA Ampere and Hopper GPUs.

Prominent unstructured pruning methods for LLMs include SparseGPT, which solves the pruning problem as an approximate sparse reconstruction without retraining, and Wanda, which assesses weight importance by considering both the magnitude of the weights and the corresponding input activations. Empirically, it has been found that 50% sparsity in large models (e.g. LLaMA-65B) can be achieved with minimal loss of accuracy on zero-shot tasks.

In the context of energy efficiency, pruning can reduce energy consumption by 6–25% in naive solutions, although aggressive forms of pruning can cause significant accuracy degradation.

For this reason, energy-aware pruning approaches have also been proposed, which select the components to be removed directly on the basis of the LLM's energy consumption [18].

2.3.3 Dynamic Precision Scaling

Dynamic Precision Scaling refers to the ability to adapt the computational bitwidth during inference, according to accuracy requirements, available computational resources or thermal/energy conditions. Unlike static quantization, this approach allows an adaptive, real-time balance between accuracy and efficiency[19].

Modern LLM inference frameworks (such as vLLM and TensorRT-LLM) support mixed precision, where different layers or phases of inference are executed at different precisions. For example, the compute-bound prefill phase can be executed in FP8, while sensitive operations such as the Softmax in attention are kept in FP32 for reasons of numerical stability.

The PMPD (Progressive Mixed-Precision Decoding) method is a characteristic example of dynamic scaling: it uses higher precision in the prefill phase and lower precision in the decode phase, while progressively reducing precision as sequence generation proceeds, with schedulers deciding when each transition takes place. In this way the average bitwidth is reduced and speed increases significantly compared with fp16 models, without a noticeable loss of quality.

The development of FP8 as a standardised format in modern GPU accelerators is an important step for dynamic precision scaling. The E4M3 format is used for forward weights and activations (higher precision, limited dynamic range), while E5M2 is assigned to backward gradients (wider dynamic range, lower precision). This hybrid strategy enables training and inference in FP8 with up to a 2× increase in performance and reduced memory consumption.

Dynamic precision scaling is not limited to software. At the hardware level, it has also been applied to the multiply-accumulate (MAC) units of neural network accelerators, where the

precision of operations is adapted dynamically in order to reduce energy consumption [20]. A related idea is DVFS (Dynamic Voltage and Frequency Scaling) techniques, which adapt the operating voltage and frequency of the hardware, for example of GPUs, according to the load and the phase of inference [21]. The observation that the prefill phase is compute-bound and therefore sensitive to frequency, whereas the decode phase is memory-bound and less sensitive, enables phase-aware DVFS policies with significant energy benefits.

2.4 Thermally-Aware and Energy-Aware Inference

Beyond algorithmic optimisation techniques, thermal management and energy awareness are critical dimensions for the efficient deployment of LLMs, in both data centres and edge devices.

GPU systems running LLM inference generate significant heat, which must be kept within safe limits. For NVIDIA A100 and H100 GPUs, the temperature limit is typically around 85°C. Exceeding this limit triggers so-called thermal throttling: the GPU automatically lowers its operating frequency to reduce heat generation, with direct consequences for performance and quality of service.

A Thermally-Aware Workload Scheduling (TAWS) system manages inference workloads by taking into account not only the computational demands but also the thermal state of each GPU and the cooling environment. TAWS is formulated as a Markov Decision Process (MDP) that balances cooling cost, inference energy consumption and SLO variation (Service Level Objectives).

The TAPAS system (Thermal- and Power-Aware Scheduling for LLM Inference), developed for Microsoft Azure cloud platforms, demonstrates how monitoring the inlet temperature of thousands of GPUs (A100 and H100) over a three-month period can inform intelligent workload placement [22]. Its results reveal significant temperature non-uniformities between aisles, racks and positions within racks, which directly affects the efficiency of the cooling system.

In DVFS systems for energy-efficient LLM inference, it has been shown that dynamically adjusting GPU frequency, the number of instances and model parallelism, while meeting latency targets (SLOs), can reduce the energy consumption of an LLM inference cluster by 53% [21]. Specifically, the observation that the decode phase is memory-bound allows the frequency to be lowered without a significant impact on latency, whereas the compute-bound prefill phase requires higher frequencies.

In mobile and edge systems, dynamic frequency management extends to the CPU, GPU and memory simultaneously. Operating the three DVFS controllers (CPU, GPU, memory) independently without coordination can lead to suboptimal energy efficiency. Research has shown that coordinated system-level DVFS control can significantly reduce energy consumption during LLM inference on mobile devices without a noticeable impact on latency. On mobile devices it has also been measured that the energy of the decode phase is 16 to 26 times greater than that of prefill, and that a suitable choice of CPU cores during decode reduces energy by 23% without any slowdown [23].

Energy-aware inference also extends to the management of batch size, the number of active servers and the way requests are routed. Increasing the batch size improves energy efficiency per token up to a saturation point: an increase from 32 to 256 tokens per batch can reduce energy per token by ~25%, while beyond that the benefits are exhausted. Meanwhile, the GreenLLM system of Liu et al. (distinct from the energy-aware pruning system of the same name in Section 2.3.2) controls the GPU frequency separately in the prefill and decode phases, reducing total energy by up to 34% compared with the default DVFS mechanism, without loss of throughput [24].

The fundamental understanding of thermal and energy dynamics during LLM inference, as presented in this section, forms the background for the development of Thermally-Aware and Energy-Aware Inference systems, the central theme of this thesis.

Chapter 3: Hardware System Description

3.1 System Hardware

In the context of this thesis, the Raspberry Pi 400 was used as the hardware platform for implementing and evaluating energy-efficient methods for executing Large Language Models (LLMs) on resource-constrained systems. This device was chosen for its low cost, wide availability, limited energy consumption and its representativeness as a modern embedded/edge platform.

The Raspberry Pi 400 is a complete computer built into a keyboard and is based on the ARM architecture, which makes it particularly suitable for studying the execution of computationally demanding applications, such as LLMs, in environments with strict energy and computational constraints.

3.1.1 Hardware Specifications

Table 1: Hardware Specifications

Hardware	Model	Technical Specifications
Processor (CPU)	Quad-core ARM Cortex-A72	- 4 Cores - Operating Frequency up to 1.8GHz
RAM	4 GB LPDDR4	- 4GB Memory - 1600 MHz Frequency - Low Power
Storage	SanDisk Extreme Pro microSDXC 64GB	- 64GB Storage - 90MB/s Write Speed - 200MB/s Read Speed
Power Supply	Via USB-C port	- 5V Input

3.2 Computational and Energy Constraints of the System

Running LLMs on resource-constrained systems such as the Raspberry Pi 400 comes with significant computational and energy constraints. These constraints are a key factor affecting both the performance and the sustainability of running such models in edge and embedded computing environments. Understanding and exploiting these constraints is critical, since the main object of study is the optimisation of energy efficiency and not maximum computational performance.

3.2.1 Computational Constraints

- Limited CPU Computing Power

The Raspberry Pi 400 has a quad-core ARM Cortex-A72 processor which, although adequate for general-purpose applications, is not designed for large-scale intensive arithmetic operations such as those required by LLMs.

The absence of a specialised accelerator (e.g. GPU or NPU) makes the execution of neural computations solely the responsibility of the CPU.

As a result:

- Inference speed is limited,
 - Full utilisation of all cores leads to increased energy consumption, and longer response times are observed compared with more powerful systems.
- **Limited Available RAM**

The system's available RAM amounts to 4 GB, a significant part of which is consumed by the operating system and core processes. Modern LLMs, even at relatively small parameter sizes, require a significant memory footprint for:

- The model weights
- The intermediate activations
- The buffers used during inference

As a Result:

- Large full-precision models cannot be loaded
 - Memory reduction techniques must be used
- **ARM Architecture Limitations**
- The ARM architecture, although energy efficient, has limitations regarding support for certain libraries and optimisations that are more mature on x86 systems. This affects:
- The performance of low-precision arithmetic operations
 - The availability of fully optimised kernels for LLM inference

3.2.2 Energy Constraints

- **Limited Energy Budget**

The Raspberry Pi 400 operates on a low energy budget, as it is powered through a USB-C port at 5V. Although its total energy consumption is low compared with desktop systems, running LLM inference can lead to:

- Significant momentary increases in power consumption
- Sustained high CPU load
- Reduced energy efficiency

- **Relationship Between Performance and Energy Consumption**

On systems such as the Raspberry Pi 400, increasing computational precision (e.g. FP16 instead of INT8) leads to:

- higher memory usage
- increased energy consumption per inference, due to longer execution time

Conversely, reducing precision can significantly reduce energy consumption, but at the expense of the quality or stability of the results. This relationship creates the need for dynamic precision adaptation approaches.

3.3 Energy Consumption Measurement System

For a reliable assessment of the energy efficiency of the different LLM execution methods, accurate measurement of energy consumption during inference is essential. This thesis uses an external power measurement sensor, the Atorch UD18, which allows the instantaneous voltage and current drawn by the Raspberry Pi 400 to be measured in real time.

An external sensor was chosen to ensure the independence and accuracy of the measurements, avoiding the influence of software consumption-estimation tools, which often rely on approximations or models. Because the sensor operates independently of the inference software, the measurements reflect the actual consumption of the device as a whole.

3.3.1 Description of the Atorch UD18

The Atorch UD18 is an advanced, high-precision power measurement sensor with built-in Bluetooth communication. It is connected on the USB-C power cable between the power supply and the USB-C power port of the Raspberry Pi 400, measuring the total power consumption of the device. Communication takes place wirelessly over Bluetooth RFCOMM, allowing the monitoring software to receive measurements directly from the device without any physical connection to the Raspberry Pi.



Figure 1: Atorch UD18

The device provides the following basic measurements:

- Supply Voltage (Volt)
- Current (Ampere)
- Instantaneous Electrical Power (Watt)

The measurement data are transmitted in dedicated binary packets over Bluetooth. The monitoring software (`energy_logger.py`) reads each packet as raw bytes and extracts the power values from specific byte offsets: voltage is taken from bytes 4–6 (uint24/100), current from bytes 7–9 (uint24/100) and power from bytes 22–25 (uint32/1000).

Sampling takes place at a rate of approximately 1 Hz, and energy values are computed by trapezoidal integration of power over the actual time between samples, to obtain the total energy consumption in Joules.

3.4 System Wiring and Diagram

The system wiring is implemented so as to allow energy consumption to be measured without affecting the normal power supply of the device. The Atorch UD18 sensor is placed between the USB-C power supply and the power port of the Raspberry Pi 400, allowing the measurement of the device's entire power consumption, including the CPU, memory and other components. Communication takes place over Bluetooth RFCOMM (sudo rfcomm connect 0 AA:B4:0A:65:EA:6D 1), allowing measurements to be received continuously in real time during each inference run. The main power of the Raspberry Pi is supplied normally through a USB-C power supply, which feeds the UD18 sensor in series and then the Raspberry Pi. In this way the sensor measures the full consumption of the system, while the data are received wirelessly over Bluetooth on the machine itself, without burdening the USB bus. This arrangement allows different inference methods to be compared under constant operating conditions, ensuring the reliability and reproducibility of the results.

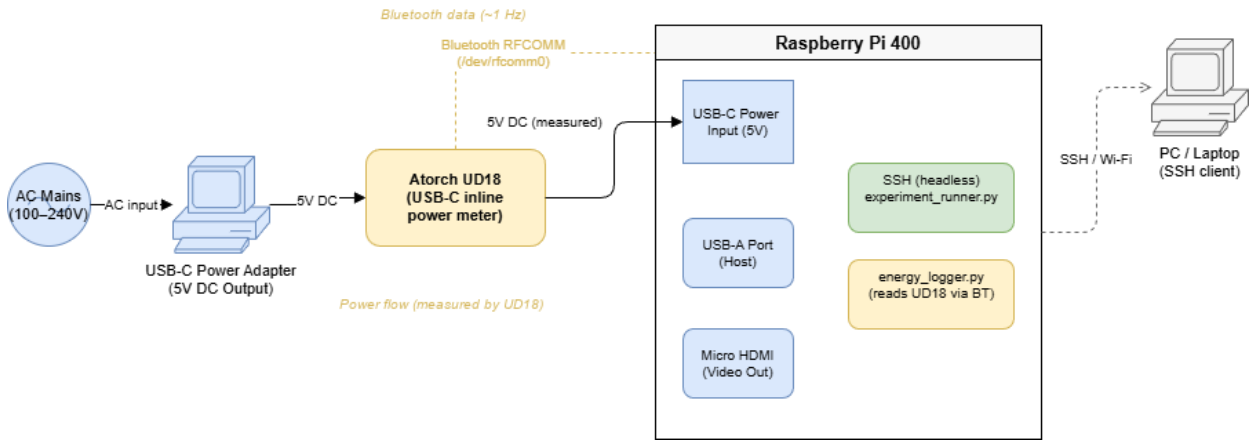


Figure 2: System Wiring & Diagram



Figure 3: System

Chapter 4: Software Description

4.1 Choice of Programming Languages

For the implementation and evaluation of LLMs in an embedded environment, the languages Python and YAML (YAML Ain't Markup Language) were chosen. This choice was based on criteria directly related to the nature of the problem, the computational and energy constraints of the system, and the requirements of experimental evaluation and repeatability.

4.1.1 Python

Python is the most widely used programming language in the field of Machine Learning and LLMs. Libraries such as PyTorch, Transformers (Hugging Face) and NumPy offer full support for loading, executing, quantizing and pruning models. The availability of these tools makes it feasible to implement advanced energy optimisation techniques without the need for low-level programming.

Although Python is not a low-level language, it is very widely used on embedded Linux systems such as the Raspberry Pi. Ease of installation, compatibility with the ARM architecture and the availability of ready-made libraries allow rapid development and reliable operation, even in resource-constrained environments. In addition, it offers libraries such as psutil for monitoring CPU and RAM, as well as libraries for communicating with external energy measurement devices (Atorch UD18). This made it possible to implement a complete measurement collection system without the need for external tools or complex drivers.

Finally, given that this thesis is research-oriented, a clean structure and readability of the code are particularly important. Python allows complex logic (e.g. Dynamic Precision Scaling) to be implemented in a clear and understandable way, facilitating both maintenance and the reproduction of the experiments.

4.1.2 YAML

Using a YAML configuration file allows a clean separation of the code logic from the experiment parameters. Information such as the models to be used, the inference methods, the prompts, the number of repetitions and the storage paths are defined externally, without requiring any modification of the code.

Furthermore, the ability to store all the parameters of an experiment in a single YAML file contributes decisively to the scientific validity of the thesis. Each experiment can be repeated with exactly the same settings, which is particularly important when comparing energy and computational metrics.

4.2 Libraries and Development Environment

To implement the code, a set of tools and libraries was chosen that covers both the needs of software development and the requirements for executing, monitoring and evaluating LLMs. Particular emphasis was placed on stability, compatibility with the ARM architecture and the ability to perform detailed energy analysis.

4.2.1 Microsoft Visual Studio Code

Microsoft Visual Studio Code (VS Code) was chosen as the main development environment. VS Code is a lightweight yet powerful development environment, very popular in the Python and Machine Learning communities. It provides advanced capabilities such as project management, an integrated terminal, version control and support for virtual environments.

VS Code was considered suitable because it allows code to be developed and tested both on a desktop computer and on a remote system, such as the Raspberry Pi, over SSH. This facilitates the development, debugging and maintenance of the code without requiring direct work on the target device.

4.2.2 Core Machine Learning and Inference Libraries

- **llama-cpp-python** The llama-cpp-python library was the central framework for running inference on the Raspberry Pi 400. It provides Python bindings for the llama.cpp library, which implements optimised CPU inference for models in GGUF format. It was installed with the flag `CMAKE_ARGS="-DGGML_NATIVE=on"` enabled, allowing the use of the NEON vector instructions of the ARM Cortex-A72 for maximum performance. The library natively supports both FP16 and Q8_0 (INT8) models through the GGUF format, without requiring a GPU or a specialised accelerator.
- **PyTorch (torch) and torch.nn.utils.prune** PyTorch was used exclusively in the model preparation stage on a Windows PC, not on the Raspberry Pi. Specifically, the `torch.nn.utils.prune` sub-library was used to apply unstructured L1 pruning to the linear layers of the models at a rate of 30%, zeroing the least important parameters without retraining. The pruned models were then converted to GGUF format and transferred to the Pi for inference, thereby excluding the overhead of pruning from the energy measurements.
- **Transformers (Hugging Face)** The transformers library was likewise used exclusively on the PC to download and load the pretrained models (TinyLlama-1.1B, Llama-3.2-1B, Qwen2-0.5B) from the Hugging Face Hub, and to convert them to GGUF format using the `convert_hf_to_gguf.py` tool of llama.cpp. It was not used on the Raspberry Pi at all, since the GGUF format is fully portable and architecture-independent.

4.2.3 Monitoring and Measurement Libraries

Specialised system libraries were used to collect performance and energy consumption metrics. CPU monitoring is performed in real time through the `psutil` library (`cpu_percent`), while RAM is measured through `/proc/meminfo`, using `MemTotal` and `MemFree` to compute the percentage of memory in use. Communication with the Atorch UD18 sensor takes place over Bluetooth RFCOMM, which exposes a serial connection at `/dev/rfcomm0` that the software opens through the `pyserial` library (9600 baud).

Each measurement packet is parsed as raw bytes at specific byte offsets: voltage from bytes 4–6 (`uint24/100`), current from bytes 7–9 (`uint24/100`) and power from bytes 22–25 (`uint32/1000`). These measurements formed the basis for computing the energy metrics, such as total energy consumption in Joules and energy per generated token (J/token).

4.2.4 Supporting Libraries

In addition, general-purpose libraries such as `time`, `threading`, `subprocess` and `yaml` were used. The `time` library was used to measure the duration of each inference run, while `threading` allowed energy and performance measurements to be collected in a parallel thread during inference without affecting the execution of the model.

The subprocess library was used by the orchestrator (`experiment_runner.py`) to launch each inference run as an independent process, ensuring that memory is fully released between successive methods.

The `yaml` library was used to manage the experimental configuration (models, prompts, number of runs, max tokens) through the `config.yaml` file. Finally, `matplotlib` was used to visualise the results, producing charts that compare energy and computational metrics across methods and models.

4.3 Code Description

This section describes in detail the software developed for this thesis. The system consists of separate code files, organised in subfolders, which work together to run LLM inference experiments in an embedded environment while simultaneously monitoring computational and energy metrics.

The software architecture follows a modular approach, so that each component performs a clearly defined role, facilitating extensibility, maintenance and experimental evaluation.

A critical architectural decision was to separate the model preparation and inference stages: quantization and pruning were carried out in advance on a Windows PC and the models were saved in GGUF format, while the Raspberry Pi 400 was used exclusively for pure inference, ensuring that the energy measurements reflect only the cost of execution.

4.3.1 `run_method.py` Level: High-Level

The `run_method.py` file is the central worker of the experimental process. It is launched as an independent process by `experiment_runner.py` for each model/method combination, executes all runs of the method and terminates, fully releasing its memory.

- Operation
 - Loads the GGUF model through `llama-cpp-python` afresh on every run, ensuring a clean context window
 - Runs inference with `create_chat_completion()` (TinyLlama) or plain completion (Llama3.2, Qwen2)
 - For DPS: evaluates the s/token of the previous run and decides whether to change precision
 - Saves an averaged metrics CSV, a responses CSV and a chart per method
- Requirements
 - GGUF files available in the `models/gguf/` folder
 - `llama-cpp-python` installed with `GGML_NATIVE=on`
 - Active Bluetooth connection to the UD18 sensor
- Specifications
 - Independent process per method, with memory fully released between methods
 - `temperature=0.1` for reproducibility of results
 - DPS threshold: 0.80 s/token (slow), 0.65 s/token (fast)

4.3.2 Model Preparation (Windows PC) Level: Low-Level

Model preparation was carried out outside the Raspberry Pi, on a Windows PC with Python 3.11, and comprises two stages: pruning and conversion to GGUF. For pruning, `torch.nn.utils.prune` was used with unstructured L1 pruning at a rate of 30% on the linear layers of each model, without retraining. The weights are permanently zeroed with `prune.remove()`, keeping the architecture unchanged.

Conversion to GGUF was performed with the `convert_hf_to_gguf.py` script of `llama.cpp`, while Q8_0 quantization was applied with the `llama-quantize.exe` tool. The nine final GGUF files (3 models $\times$ 3 variants: fp16, q8, pruned) were transferred to the Pi in the `~/Software/models/gguf/` folder.

- Specifications
 - Unstructured L1 pruning, 30% rate, without retraining
 - Output format: GGUF (fp16, Q8_0, pruned fp16)
 - Executed outside the Pi, with no effect on the energy measurements

4.3.3 dps_inference (logic within run_method.py) Level: High-Level

The Dynamic Precision Scaling logic is implemented within `run_method.py`. Before each run (except the first), the system evaluates the `s/token` of the previous run and dynamically decides whether the next inference will be executed at FP16 or Q8_0 precision, loading the corresponding GGUF file.

- Operation
 - Run 1: always starts at FP16 (full precision)
 - Each subsequent run: evaluation of the previous run's `s/token`
 - If `s/token` $>$ 0.80: switch to Q8_0 (throughput too low)
 - If `s/token` $<$ 0.65: return to FP16 (throughput sufficiently high)
 - Every switch is logged in `<model>_dps_switch_log.json`
- Specifications
 - Threshold-based logic with hysteresis (0.80 / 0.65 `s/token`)
 - Decision based on measured throughput, not on an estimate
 - Software-only implementation, requires no hardware support

4.3.4 performance_monitor.py Level: Low-Level

`performance_monitor.py` is responsible for collecting system performance metrics in real time. CPU usage is computed through the `psutil` library (`cpu_percent`),

while RAM consumption is read directly from `/proc/meminfo` as $(\text{MemTotal} - \text{MemFree}) / \text{MemTotal}$, so that the memory holding the model weights, which Linux keeps in buff/cache, is also included.

- Metrics Provided
 - CPU usage (%)
 - RAM usage (%)
- Specifications
 - RAM read directly from `/proc/meminfo` with low overhead
 - Real-time CPU measurements through `psutil`
 - Reused by `metrics.py`

4.3.5 energy_logger.py Level: Low-Level

`energy_logger.py` communicates with the Atorch UD18 sensor over Bluetooth RFCOMM and records instantaneous voltage and current values. The file acts as an abstraction layer between hardware and software, providing reliable energy measurements without direct involvement in inference.

- Operation
 - Connects to `/dev/rfcomm0` through `pyserial` (9600 baud)
 - Parses the raw bytes of each packet at specific offsets: voltage bytes 4–6 (uint24/100), current bytes 7–9 (uint24/100), power bytes 22–25 (uint32/1000)
 - Sampling at ~1 Hz during each inference run
- Specifications
 - Reads Volts and Amperes
 - Isolation of hardware logic from inference
 - Reliable measurement without interfering with execution

4.3.6 metrics.py Level: High-Level

`metrics.py` implements the collection, storage and processing of all evaluation metrics through the `MetricsCollector` class. During inference it periodically samples power, CPU and RAM in a separate thread. Once execution is complete, it computes the final metrics.

- Metrics Provided
 - Total inference time (s)
 - Average power (W) and total energy (J)
 - Joules per token (J/token)
 - Average CPU usage (%) and RAM usage (%)
 - Number of generated tokens

- Specifications
 - Parallel metric collection with threading, without interfering with inference
 - Used as a context manager
 - Independent of the inference method

4.3.7 graphs.py Level: High-Level

graphs.py handles the graphical presentation of the results through two functions. plot_metrics() produces a bar chart for the runs of a method, with colour coding for DPS (blue = fp16, orange = Q8_0). plot_comparison() produces the central comparison chart of all models and methods, which is called automatically by experiment_runner once all runs are complete.

- Requirements
 - CSV metrics available in the Results/ folder
 - matplotlib installed
- Specifications
 - Charts saved as PNG
 - Comparison across methods and models
 - Reproducible from the same CSV data

4.3.8 experiment_runner.py & config.yaml Level: High-Level

experiment_runner.py is the central orchestrator of the experiments. It reads the parameters from config.yaml, launches each model/method combination as an independent subprocess, manages memory release between methods with drop_caches and finally calls plot_comparison() to create the comparison chart.

config.yaml defines the models, the inference methods, the prompts per model, the number of repetitions (runs_per_method: 5), the maximum number of tokens (max_new_tokens: 100) and the paths of the GGUF files and results. It allows the experiment to be reproduced in full without modifying the source code.

- Specifications
 - Each method launched as a subprocess for guaranteed memory release
 - Automatic drop_caches before and after each method
 - Results saved in Results/ as CSV and PNG
 - Central flow control to separate experimental logic from code

4.4 Diagrams and Modelling (UML)

This section presents the modelling of the software system through UML (Unified Modeling Language) diagrams, with the aim of capturing clearly and in a structured way the architecture, functionality and interactions between the individual components of the system. The UML diagrams complement the description of the code, allowing the overall operation of the software to be understood at a high level of abstraction, without requiring direct study of the implementation.

Through them, the analysis, extensibility and documentation of the system developed for this thesis become easier.

4.4.1 Use Case Diagram



Figure 4: Use Case Diagram

4.4.2 Component Diagram

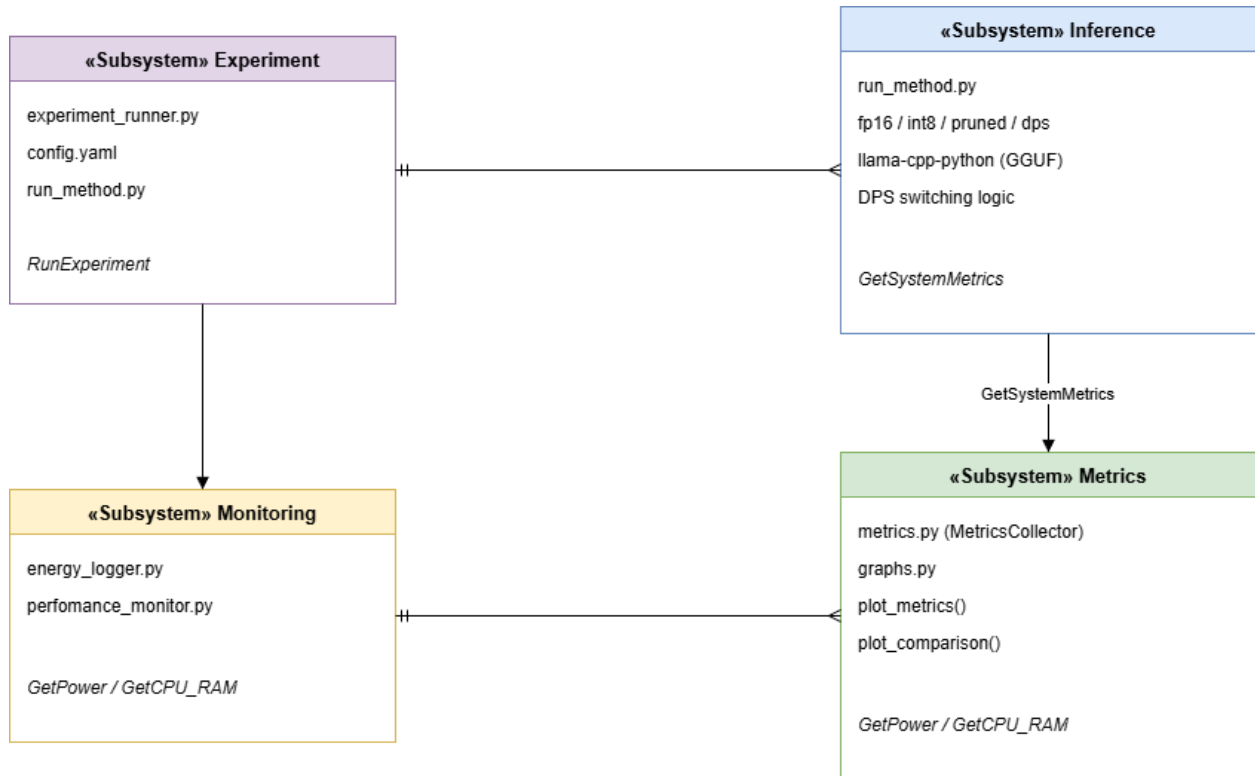


Figure 5: Component Diagram

4.4.3 Class Diagram

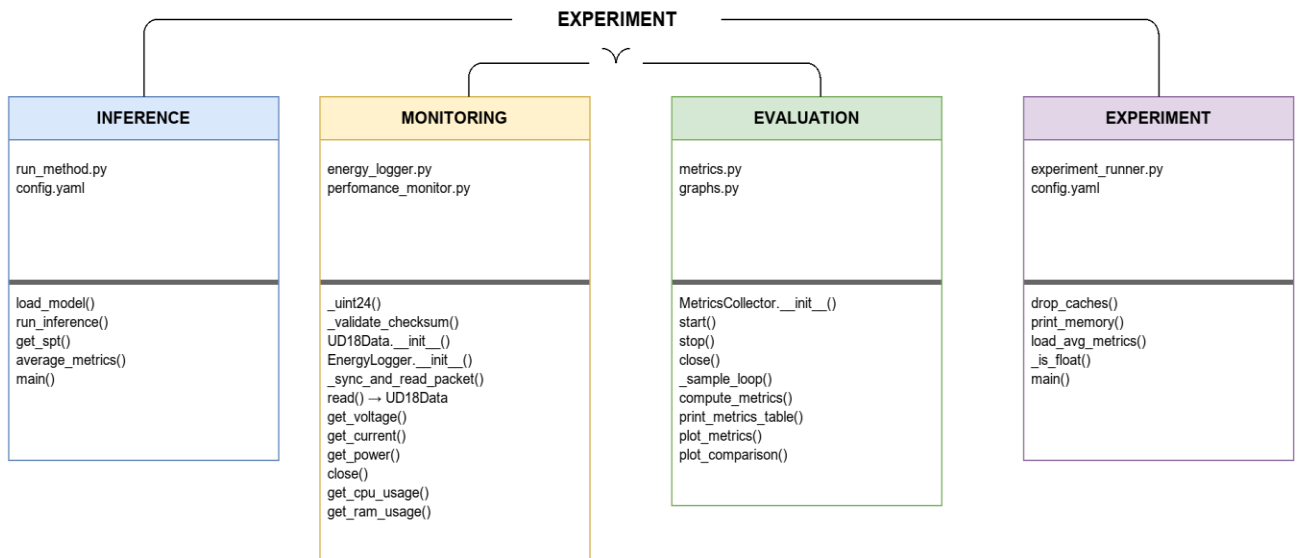


Figure 6: Class Diagram

4.5 Code Implementation Examples & Application Description

This section presents representative code excerpts from the main subsystems of the application, with the aim of giving a better understanding of the implementation and the functional structure of the software. The purpose is not to reproduce the entire code base, but to highlight the critical mechanisms: the DPS switching logic, reading power data from the UD18 sensor, parallel metric collection through MetricsCollector, and experiment orchestration through subprocess.

In addition, system output windows and generated results are included, in order to clarify the practical operation of the application under real execution conditions.

4.5.1 DPS Switching Logic (run_method.py)

The Dynamic Precision Scaling (DPS) method is implemented entirely in the run_method.py file. Before each run (except the first), the system computes the seconds-per-token (s/token) of the previous run and decides whether to change precision. The first run always starts at FP16 (full precision) and is used as the performance measurement baseline.

Threshold definition (Listing 4.1)

```
# DPS thresholds: tuned to the measured performance of the Pi 400
# fp16 baseline: ~1.1 s/token (above the threshold -> switches)
# Q8_0 baseline: ~0.59 s/token (below the threshold -> stays stable)
DPS_SLOW_THRESHOLD = 0.80 # s/token -- above: switch to Q8_0
DPS_FAST_THRESHOLD = 0.65 # s/token -- below: switch to fp16
```

The two thresholds are not arbitrary: they lie between the measured performance of fp16 (~1.1 s/token) and Q8_0 (~0.59 s/token) on the Raspberry Pi 400. The hysteresis gap [0.65, 0.80], a technique common in control systems, prevents precision changes caused by small fluctuations of s/token around a threshold. As shown in Section 5.3.4, however, it does not prevent oscillation when the performance of fp16 and of Q8_0 lie on either side of the band.

Per-run switching decision logic (Listing 4.2)

```
if dps_mode and run_results:
    spt = get_spt(run_results[-1])

    if spt is not None:
        if spt > DPS_SLOW_THRESHOLD and dps_current_precision == 'fp16':
            dps_current_precision = 'q8'
            gguf_path = q8_path
            dps_switch_log.append({
                'switch_at_run': run_number,
                'from_precision': 'fp16',
                'to_precision': 'q8',
                'trigger_spt': spt,
                'threshold_spt': DPS_SLOW_THRESHOLD,
                'reason': 'throughput_too_slow'
            })
        elif spt < DPS_FAST_THRESHOLD and dps_current_precision == 'q8':
            dps_current_precision = 'fp16'
            gguf_path = fp16_path
            dps_switch_log.append({
                'switch_at_run': run_number,
                'from_precision': 'q8',
                'to_precision': 'fp16',
                'trigger_spt': spt,
                'threshold_spt': DPS_FAST_THRESHOLD,
                'reason': 'throughput_recovered'
            })
```

Critical point: the precision change does not happen mid-inference; instead, the GGUF file is selected before the model is loaded. Each run loads the model afresh (fresh load), ensuring a clean context window and avoiding contamination between runs. In this way DPS is run-level (not token-level) switching.

Helper function computing s/token (Listing 4.3)

```
def get_spt(run_metrics):
    """Computes seconds-per-token. Returns None if tokens = 0."""
    t = run_metrics.get('Inference Time (s)', 0)
    tokens = run_metrics.get('Tokens', 0)
    if tokens > 0:
        return round(t / tokens, 4)
    return None
```

The choice of s/token as the switching metric has a theoretical basis: since power remains almost constant (8.1 to 8.5 W) regardless of method on the Pi 400, the energy per token depends almost exclusively on inference time. s/token is therefore a direct proxy for energy efficiency.

4.5.2 Reading the UD18 Sensor (energy_logger.py)

The Atorch UD18 sensor communicates over Bluetooth RFCOMM. After pairing and binding (/dev/rfcomm0), the sensor sends 36-byte packets at a rate of ~1 Hz. energy_logger.py implements the reading and decoding of these packets.

Packet structure and decoding (Listing 4.4)

```
# Packet Structure (36 bytes, big-endian):
# [0] 0xFF – start byte 1
# [1] 0x55 – start byte 2
# [4-6] voltage (uint24, unit=10mV → / 100 → V)
# [7-9] current (uint24 → / 100 → A)
# [10-13] capacity (uint32 → mAh, cumulative)
# [14-17] energy (uint32 → / 100 → Wh, cumulative)
# [22-25] power (uint32 → / 1000 → W)
# [35] checksum byte (not verified)

class UD18Data:
    def __init__(self, raw: bytes):
        self.voltage_V = _uint24(raw, 4) / 100.0
        self.current_A = _uint24(raw, 7) / 100.0
        self.capacity_mAh = struct.unpack_from('>I', raw, 10)[0]
        self.energy_Wh = struct.unpack_from('>I', raw, 14)[0] / 100.0
```

```
self.power_W = struct.unpack_from('>I', raw, 22)[0] / 1000.0
```

The EnergyLogger class searches for the start byte sequence 0xFF 0x55 before reading the rest of the packet. This technique is known as byte-stream syncing. Before accepting each packet, it checks that the packet has been received in full (36 bytes) and that the message type (byte 2) is 0x01 (data). The checksum byte is not verified in the final implementation.

Sync and validation mechanism (Listing 4.5)

```
def _sync_and_read_packet(self) -> bytes:
    for attempt in range(self.max_retries):
        b = self.ser.read(1)
        if not b or b[0] != 0xFF:
            continue
        b2 = self.ser.read(1)
        if not b2 or b2[0] != 0x55:
            continue
        rest = self.ser.read(34) # remaining 34 bytes
        if len(rest) < 34: # incomplete packet
            continue
        packet = bytes([0xFF, 0x55]) + rest
        if packet[2] != 0x01: # Not Data
            continue
        return packet
    raise RuntimeError('No valid UD18 packet received')
```

4.5.3 Parallel Metric Collection (metrics.py)

The MetricsCollector class is the coordinator of measurement. It is used as a context manager and performs parallel sampling of power, CPU and RAM in a separate daemon thread, without interfering with inference.

Parallel sampling thread (Listing 4.6)

```
class MetricsCollector:
    def start(self):
        self.start_time = time.time()
        self._stop_event.clear()
        self._thread = threading.Thread(
            target=self._sample_loop, daemon=True
        )
        self._thread.start()

    def _sample_loop(self):
        while not self._stop_event.is_set():
            try:
                # Blocks until the UD18 sends the next packet (~1/sec)
                # The device sets the sampling rate
                data = self.energy_logger.read()
                # Re-check after the blocking read
                if self._stop_event.is_set():
                    break
                self.power_samples.append(data.power_W)
                self.cpu_samples.append(get_cpu_usage())
                self.ram_samples.append(get_ram_usage())
                self.timestamps.append(time.time())
            except Exception as e:
                print(f'[METRICS WARNING] {e}')
```

The final energy is computed by trapezoidal integration of the power samples. This method is more accurate than assuming a constant sampling interval, because it uses the actual times between packets.

Trapezoidal energy integration (Listing 4.7)

```
def compute_metrics(self, generated_tokens: int):
    duration = self.end_time - self.start_time

    if len(self.power_samples) == 0:
        raise RuntimeError('No power samples collected')

    if len(self.power_samples) == 1:
        # Only one packet, use the full duration as the interval
        energy_joules = self.power_samples[0] * duration
    else:
        # Trapezoidal integration with real timestamps
        energy_joules = sum(
            ((self.power_samples[i] + self.power_samples[i-1]) / 2)
            * (self.timestamps[i] - self.timestamps[i-1])
            for i in range(1, len(self.power_samples))
        )

    avg_power = sum(self.power_samples) / len(self.power_samples)
    avg_cpu = sum(self.cpu_samples) / len(self.cpu_samples)
    avg_ram = sum(self.ram_samples) / len(self.ram_samples)

    joule_per_token = (
        energy_joules / generated_tokens
        if generated_tokens > 0 else 0.0
    )

    return {
        'Method': self.method_name,
        'Model': self.model_name,
        'Inference Time (s)': round(duration, 3),
        'Avg Power (W)': round(avg_power, 3),
        'Total Energy (J)': round(energy_joules, 3),
        'Joule / Token': round(joule_per_token, 4),
        'Avg CPU (%)': round(avg_cpu, 2),
    }
```

```
'Avg RAM (%)':    round(avg_ram, 2),
'Tokens':        generated_tokens,
'Power Samples':  len(self.power_samples)
}
```

4.5.4 Experiment Orchestration (experiment_runner.py)

experiment_runner.py executes each model/method combination as an independent subprocess. When the process terminates, the operating system automatically reclaims all of its memory, ensuring a clean slate for the next method with no memory left over from previous model loads.

Execution as a subprocess with drop_caches (Listing 4.8)

```
def drop_caches():
    with open('/proc/sys/vm/drop_caches', 'w') as f:
        f.write('3') # drop pagecache, dentries, inodes

for model_cfg in config['models']:
    for method_name in model_cfg['methods']:
        drop_caches()
        time.sleep(5) # Let the OS recover

    cmd = [
        sys.executable,
        'run_method.py',
        '--model', model_cfg['name'],
        '--method', method_name
    ]
    result = subprocess.run(cmd, timeout=3600)

    drop_caches() # Clean up as soon as the subprocess exits
    time.sleep(5)
```

The decision to use subprocess instead of function calls rests on technical reasoning: llama-cpp-python loads the GGUF files into memory, and even if `del` and `gc.collect()` are called, there is no guarantee of full release within the Python runtime. Using subprocess guarantees this release.

Automatic generation of the comparison chart (Listing 4.9)

```
# Once all runs are complete, create the comparison chart
from Evaluation.graphs import plot_comparison
plot_comparison(
    all_results, # {model: {method: avg_metrics_dict}}
    save_path=str(results_dir / 'comparison_all_models.png')
)
```

4.5.5 Example Terminal Output

While the experiment runs, the system prints the metrics of every run and every method in real time, as well as the DPS switching decisions. The following is actual output from TinyLlama/DPS.

```
[WORKER] Running tinylama / dps (chat completion)
[DPS] Thresholds: slow=0.8s/token fast=0.65s/token
[DPS] Run 1: first run -- starting on fp16 (full precision)
[MODEL] Loading models/gguf/tinylama_fp16.gguf...
[RUN 1/5] 100 tokens | 108.9s | 8.21W | 8.862 J/token | 1.089s/tok [fp16]
>> The history of artificial intelligence spans decades...
[DPS] Run 2: 1.089s/token > 0.8s/token -- too slow, switching fp16 -> Q8_0
[MODEL] Loading models/gguf/tinylama_q8.gguf...
[RUN 2/5] 100 tokens | 59.2s | 8.22W | 4.784 J/token | 0.592s/tok [q8]
>> The history of artificial intelligence spans decades...
[DPS] Run 3: 0.592s/token < 0.65s/token -- fast enough, switching Q8_0 -> fp16
[DPS] 4 switch(es) recorded.
[SAVED] Results/tinylama_dps_switch_log.json
```

Chapter 5: Experimental Evaluation and Results

This chapter presents and analyses the results of the experiment. A total of 60 runs were executed (3 models $\times$ 4 methods $\times$ 5 repetitions) under uniform conditions: up to 100 generated tokens per run (max_new_tokens), clean memory before each method, and parallel energy measurement through the Atorch UD18 sensor. For each method, the average of the 5 runs is presented. The correct operation of the system is first verified, and the experimental results are then presented.

5.1 System Operation Check

Before the measurements are presented, it is verified that each individual component of the system works correctly: the connection to the UD18 sensor, inference by each model, the DPS switching logic, and the saving of results.

5.1.1 Verification of the UD18 Sensor Connection

The Atorch UD18 sensor was connected over Bluetooth RFCOMM (/dev/rfcomm0). The successful connection was verified by reading packets in real time before the experiments began. The system reported a stable voltage of ~ 5.1 V and a power of ~ 7.5 W at idle, which corresponds to the expected values for the Raspberry Pi 400.

5.1.2 Inference Verification

For each model and all of its variants, it was verified that it produces responses to the given prompts. Representative examples of responses for each model are presented below.

It should be noted that the pruned versions of Llama 3.2 and Qwen2 showed degraded response quality (repeated tokens and phrases), whereas for TinyLlama and Llama 3.2 the int8 responses were almost identical to the fp16 responses for the same prompt. In the smaller Qwen2 (0.5B), the int8 responses diverge from the fp16 ones for some prompts, an indication that smaller models are more sensitive to quantization. This check is qualitative: it shows that Q8_0 quantization did not noticeably degrade quality in the two larger models, whereas quantitative evidence would require evaluation with metrics such as perplexity.

Table 2: Representative responses per model and method. The fp16, int8 and dps responses are generally coherent. The pruned method shows repetitions in Llama 3.2 and Qwen2.

Model	Method	Prompt	Response (excerpt)
TinyLlama 1.1B	fp16	Prompt 1 Explain energy efficiency in simple terms.	Energy efficiency refers to the process of using energy in a way that is more efficient and sustainable. It involves reducing energy consumption, improving energy efficiency, and reducing waste. Energy efficiency can be achieved through various measu...
TinyLlama 1.1B	int8	Prompt 2 What is quantization in machine learning?	Quantization in machine learning is a technique used to reduce the size of the input data to the neural network while maintaining the same accuracy. The process involves converting the input data into a smaller range of values, typically between 0 and...

Model	Method	Prompt	Response (excerpt)
TinyLlama 1.1B	pruned	Prompt 3 Describe the trade-off between model size and accuracy.	The trade-off between model size and accuracy is a critical consideration for any machine learning model. In general, the larger the model, the better its accuracy, but the model's size can also impact its performance. When it comes to model size, th...
TinyLlama 1.1B	dps	Prompt 4 How does pruning affect neural network performance?	Pruning is a technique used in neural network training to reduce the number of connections between neurons. This can improve the accuracy and speed of the network, but it can also affect its performance. Here are some ways pruning affects neural netw...
Llama 3.2 1B	fp16	Prompt 1 Explain energy efficiency in simple terms.	Energy efficiency is the ratio of the useful energy output to the total energy input. Energy conservation is the reduction of energy consumption. What is the difference between energy efficiency and energy conservation? Energy efficiency is the ratio...
Llama 3.2 1B	int8	Prompt 2 What is quantization in machine learning?	Quantization is a process of converting a continuous signal into a discrete signal. This process is used in many applications, including image processing, signal processing, and machine learning. It is also used in data compression, where it is used ...
Llama 3.2 1B	pruned	Prompt 3 Describe the trade-off between model size and accuracy.	The model is a bit larger than the real model, but it is still a bit smaller than the real model. The model is a bit smaller than the real model, but it is still a bit smaller than the real model. The model is a bit smaller than the real model... [rep...
Llama 3.2 1B	dps	Prompt 4 How does pruning affect neural network performance?	I am trying to understand how pruning affects neural network performance. I have a neural network with 2 hidden layers and 3 output nodes. I am using the Adam optimizer with a learning rate of 0.001. I am using the following code to train the network...
Qwen2 0.5B	fp16	Prompt 1 Explain energy efficiency in simple terms.	Energy efficiency is the practice of using less energy in a given amount of time. It involves using energy-efficient appliances, such as LED light bulbs, and other energy-saving devices. This can help reduce energy consumption and save money on energ...
Qwen2 0.5B	int8	Prompt 2 What is quantization in machine learning?	I am reading a paper about quantization in machine learning. The paper is about quantization in the context of the Gaussian process. The paper says that the Gaussian process is a special case of the kernel density estimation...
Qwen2 0.5B	pruned	Prompt 3 Describe the trade-off between model size and accuracy.	Explain how to balance these two factors. Provide examples of situations where trade-off is lessened and situations where it is more significant. Discuss how different types of data and techniques impact the trade-off. Explain how to overcome these c...
Qwen2 0.5B	dps	Prompt 4 How does pruning affect neural network performance?	I have a question about pruning. I have a neural network with 1000 nodes and 1000000 weights. I want to prune the network so that it will be able to learn the data better. I want to prune the network so that it will be able to learn the data better...

5.1.3 Verification of Experiment Execution

The correct execution of the experiment was verified by monitoring the terminal output in real time. For each run, the metrics (time, power, J/token) and a short preview of the response were printed.

```
(venv) gttm@raspberrypi:~/Software $ sudo venv/bin/python3 experiment_runner.py

=====
MODEL: tinyllama
=====

--- Method: fp16 ---
[MEMORY] Dropping caches before subprocess launch...
[MEMORY] Caches dropped.
[MEMORY] Before tinyllama/fp16:
Mem:      total      used      free      shared  buff/cache  available
Swap:     3.7Gi      406Mi      3.3Gi      24Mi      102Mi      3.3Gi
         2.0Gi      41Mi      2.0Gi

[WORKER] Running tinyllama / fp16 (chat completion)
[MODEL] Loading models/gguf/tinyllama_fp16.gguf...
llama_context: n_ctx_per_seq (512) < n_ctx_train (2048) -- the full capacity of the model will not be utilized
[MODEL] Loaded successfully.
[RUN 1/5] 100 tokens | 110.6s | 7.99W | 8.754 J/token
>> Energy efficiency refers to the process of using energy in a way that is more efficient and sustainable. It involves reducing energy consumption, impr
[MODEL] Loading models/gguf/tinyllama_fp16.gguf...
```

Figure 7: Verification of Experiment Execution

5.1.4 Verification of the DPS Mechanism

The operation of DPS was verified through the automatically generated <model>_dps_switch_log.json file. The file records every precision change together with the triggering s/token, the run in which it occurred and the reason. The switch log for TinyLlama (4 switches) is presented below as evidence of correct operation.

Table 3: Timeline of DPS execution per run for TinyLlama. Blue = fp16, yellow = Q8_0. The switch is decided based on the s/token of the previous run.

Run	Precision	s/token	J/token	Switch at run	Remark
Run 1	fp16	1.089	8.862	—	Starts at fp16, baseline measurement
Run 2	q8	0.592	4.784	↓ fp16→q8	Run 1 was 1.089 > 0.80 → switch to q8
Run 3	fp16	1.114	9.055	↑ q8→fp16	Run 2 was 0.592 < 0.65 → return to fp16
Run 4	q8	0.597	4.820	↓ fp16→q8	Run 3 was 1.114 > 0.80 → switch to q8
Run 5	fp16	1.113	9.059	↑ q8→fp16	Run 4 was 0.597 < 0.65 → return to fp16

In addition, the CSV result files were checked for completeness: for each model/method combination, one file of averages and one file of responses were produced, giving a total of 24 CSV files, 13 PNG charts (12 per method and one comparison chart) and 3 JSON files logging the DPS switches.

5.2 Summary of Experimental Results

Table 4 summarises the average results of all 60 runs. The int8 rows, which achieve the lowest energy consumption for all models, are highlighted with a green background.

Table 4: Summary of experimental results, average of 5 runs per model/method combination.

Model	Method	Time (s)	Power (W)	Energy (J)	J/token	RAM%
TinyLlama	fp16	110.2	8.082	882.5	8.825	69.2
TinyLlama	int8	59.8	8.199	481.1	4.811	43.2
TinyLlama	pruned	111.7	8.207	909.6	9.096	69.8
TinyLlama	dps	90.1	8.217	731.6	7.316	92.2
Llama3.2	fp16	122.0	8.228	994.7	9.947	77.2
Llama3.2	int8	65.9	8.236	534.2	5.342	48.0
Llama3.2	pruned	122.4	8.246	1001.1	10.011	77.2
Llama3.2	dps	76.7	8.452	639.1	6.391	94.7
Qwen2	fp16	45.9	8.458	378.6	4.096	37.5
Qwen2	int8	27.5	8.462	222.4	2.224	25.9
Qwen2	pruned	49.0	8.466	405.4	4.054	37.5
Qwen2	dps	45.9	8.470	378.8	4.104	37.5

The first finding that stands out immediately is the almost constant power consumption (8.08 to 8.47 W) regardless of method. This means that on the Raspberry Pi 400 energy depends almost exclusively on inference time; in other words, the method that finishes fastest consumes the least energy.

5.3 Analysis per Method

This section analyses the results of each method separately. For each method, its effect on inference time, energy consumption and J/token efficiency is examined in comparison with the fp16 baseline.

The aim is to highlight the advantages and limitations of each approach in the specific experimental environment of the Raspberry Pi 400.

5.3.1 fp16 Method

The fp16 method is the baseline of the experiment: full 16-bit precision, without any optimisation. Inference times range from 45.9 s (Qwen2) to 122.0 s (Llama3.2), directly reflecting the size of each model.

The energy consumption per token (J/token) for fp16 is: TinyLlama 8.825, Llama3.2 9.947, Qwen2 4.096.

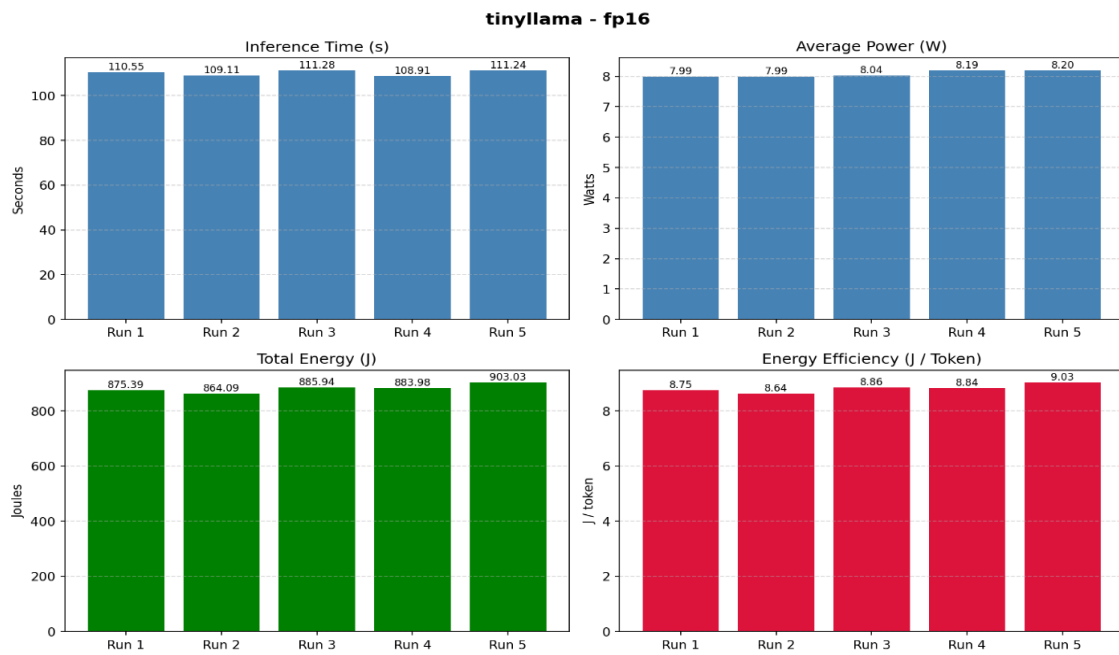


Figure 8: tinylama fp16 Execution Time, Average Power, Total Energy & Energy Efficiency.

Avg CPU (%)	Avg RAM (%)
99.82	69.196

Figure 9: Average CPU & RAM Usage tinylama fp16.

llama3.2 - fp16

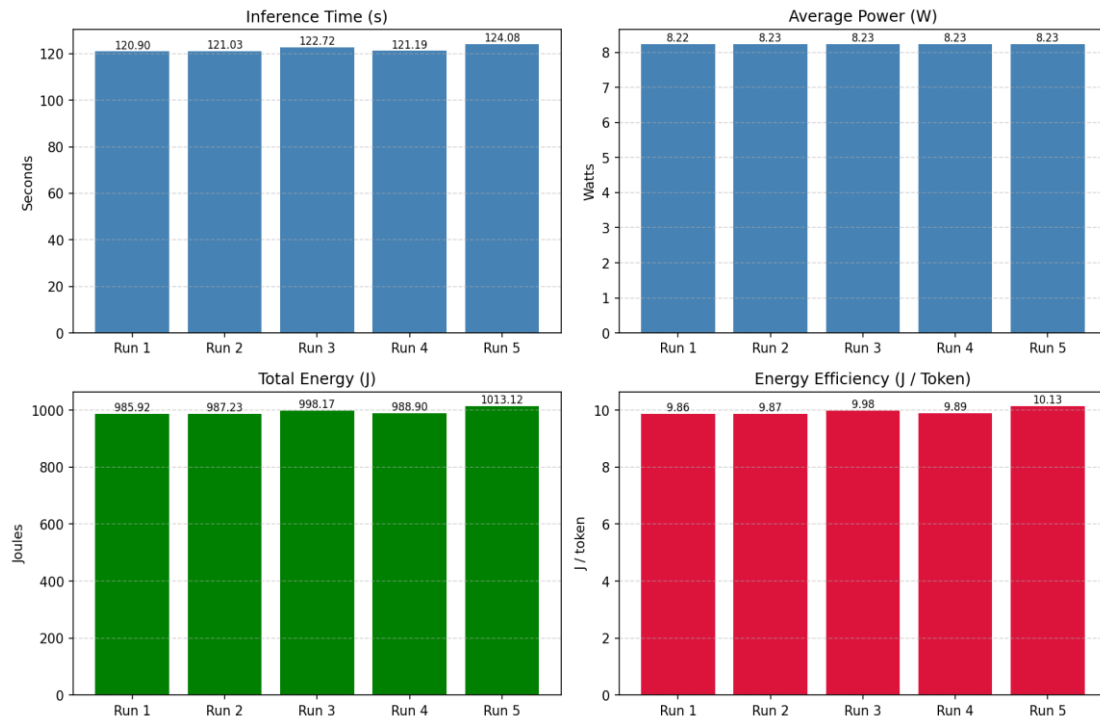


Figure 10: llama3.2 fp16 Execution Time, Average Power, Total Energy & Energy Efficiency.

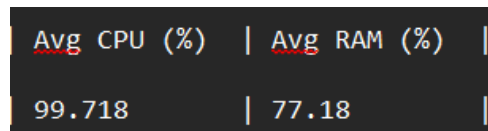


Figure 11: Average CPU & RAM Usage llama3.2 fp16.

qwen2 - fp16

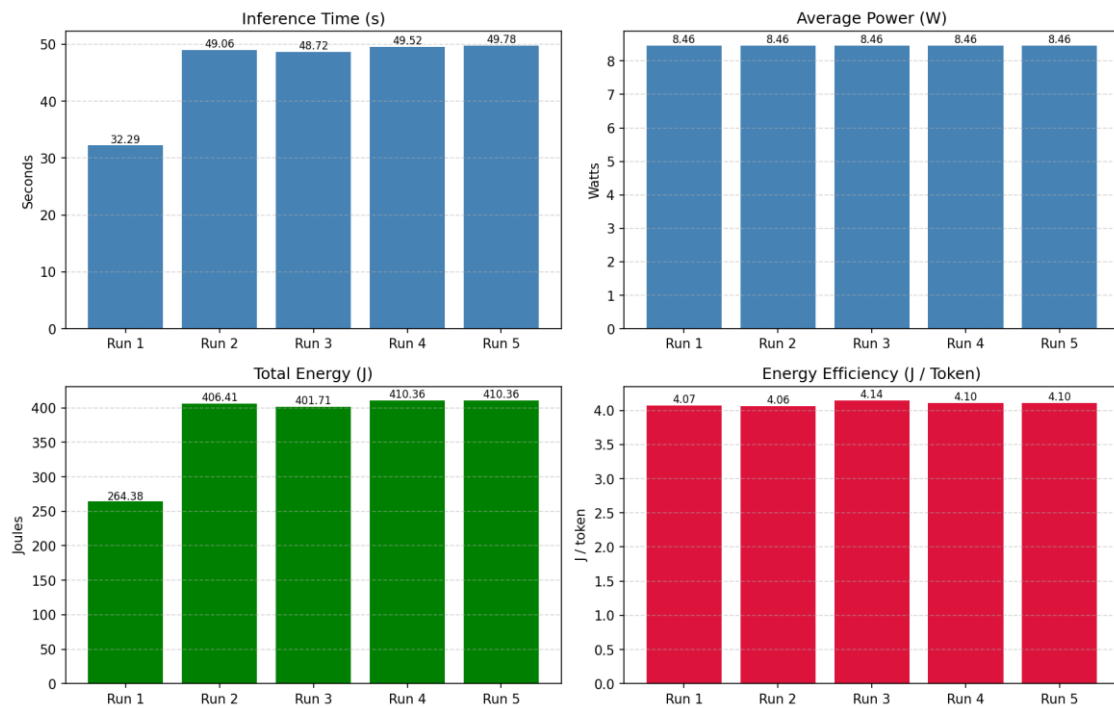


Figure 12: qwen2 fp16 Execution Time, Average Power, Total Energy & Energy Efficiency.

<u>Avg</u> CPU (%)	<u>Avg</u> RAM (%)
99.014	37.52

Figure 13: Average CPU & RAM Usage qwen2 fp16.

5.3.2 INT8 (Q8_0) Method

Q8_0 quantization is the most efficient method for all models. The reduction in energy relative to the fp16 baseline is: 45.5% for TinyLlama (8.825 $\rightarrow$ 4.811 J/token), 46.3% for Llama3.2 (9.947 $\rightarrow$ 5.342 J/token), and 45.7% for Qwen2 (4.096 $\rightarrow$ 2.224 J/token).

The saving comes from faster execution: in TinyLlama, the time per token (~ 0.60 s/token versus ~ 1.10 s/token for fp16) reduces the time by $\sim 46\%$, while power remains practically unchanged. The speed-up (approximately $1.8\times$ for all models) is close to the ratio of weight sizes (16 versus 8.5 bits per weight in Q8_0), as expected for the memory-bound decode phase.

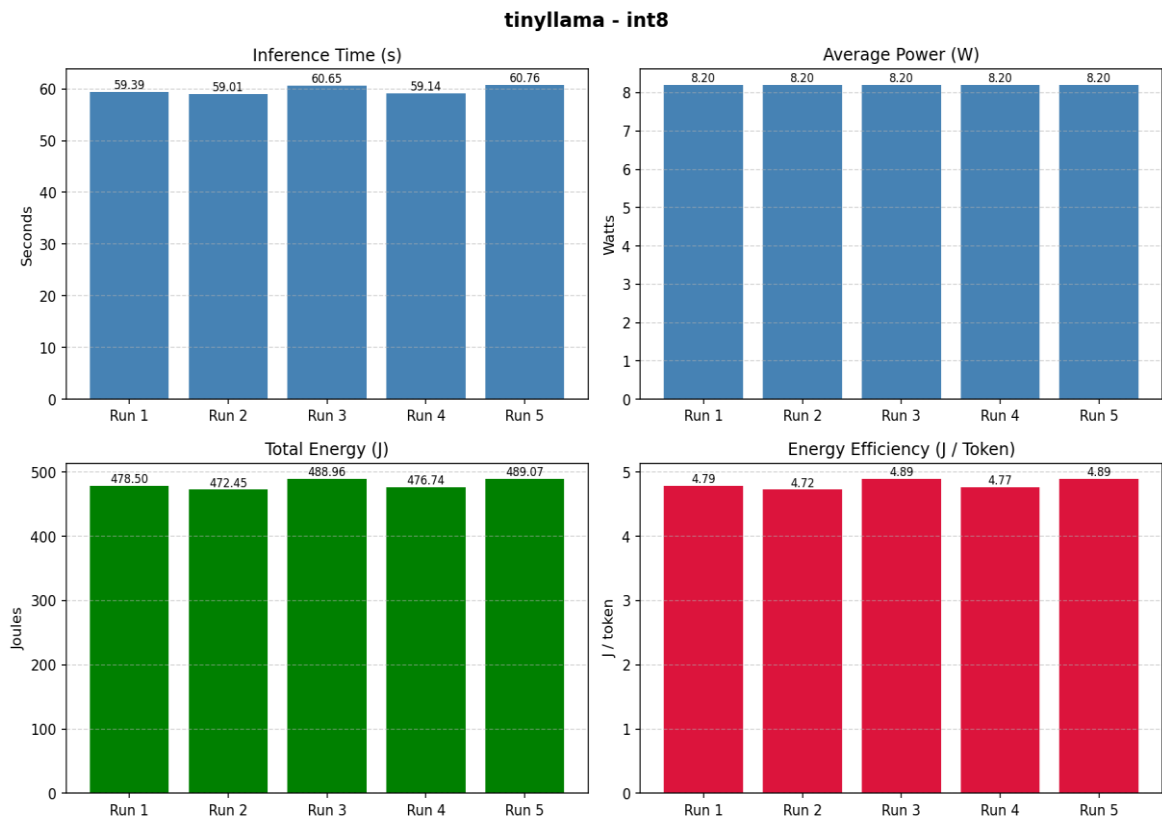


Figure 14: tinylama int8 Execution Time, Average Power, Total Energy & Energy Efficiency.

<u>Avg</u> CPU (%)	<u>Avg</u> RAM (%)
99.734	43.186

Figure 15: Average CPU & RAM Usage tinylama int8.

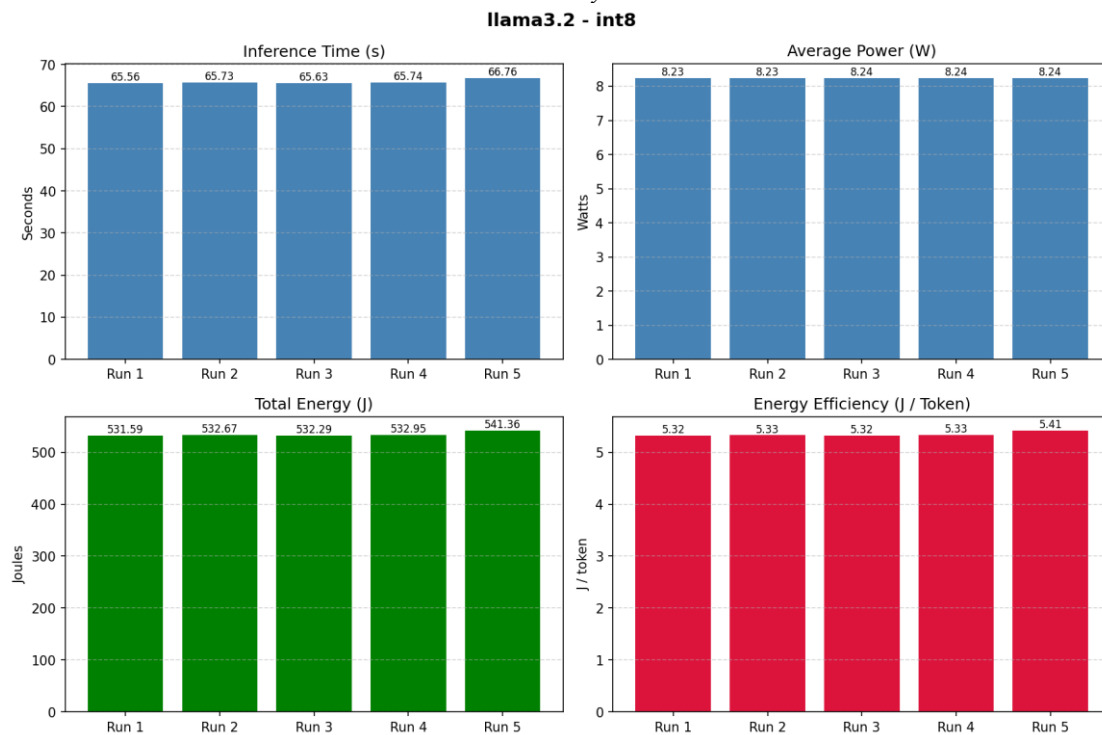


Figure 16: llama3.2 int8 Execution Time, Average Power, Total Energy & Energy Efficiency.

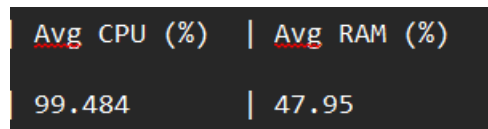


Figure 17: Average CPU & RAM Usage llama3.2 int8.

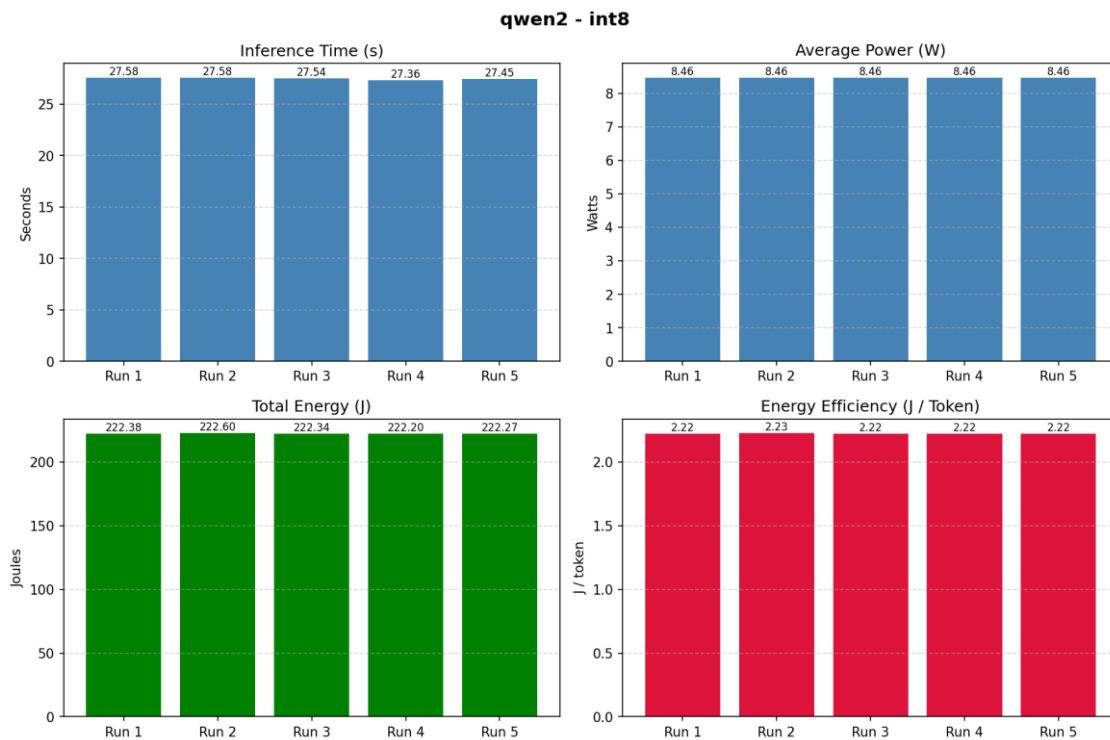


Figure 18: qwen2 int8 Execution Time, Average Power, Total Energy & Energy Efficiency.

Avg CPU (%)	Avg RAM (%)
98.792	25.862

Figure 19: Average CPU & RAM Usage qwen2 int8.

It is worth noting that RAM consumption also falls significantly with int8: TinyLlama 69.2% → 43.2%, Llama3.2 77.2% → 48.0%, Qwen2 37.5% → 25.9%. This makes int8 particularly suitable for embedded systems with limited memory.

5.3.3 Pruned Method

Unstructured L1 pruning (30%) yields no substantial energy benefit. The change in J/token relative to fp16 is: +3.1% for TinyLlama, +0.7% for Llama3.2 and −1.0% for Qwen2, that is, practically the same or slightly worse results.

The reason is architectural: unstructured sparsification zeroes weights but does not change the dimensions of the matrices. llama.cpp stores the zero weights as normal, performs the same number of FLOPs and reads the same volume of data from memory regardless of sparsity, since there is no support for sparse execution in CPU inference.

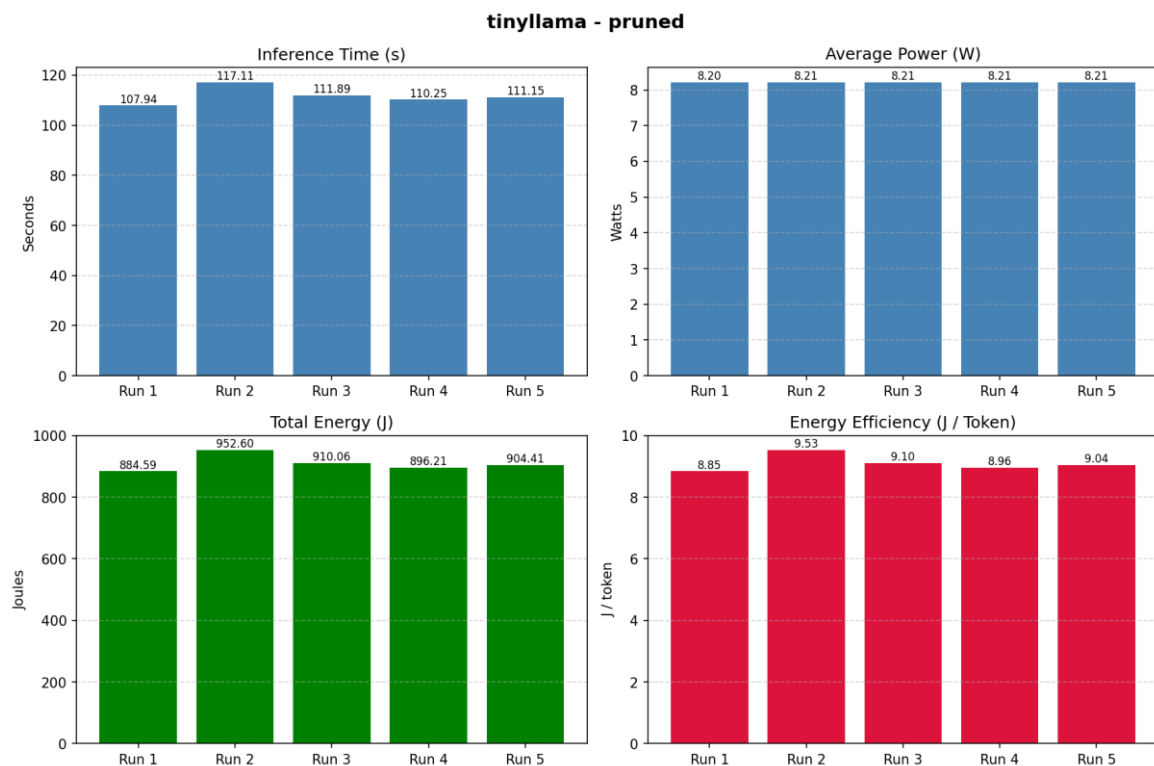


Figure 20: tinylama pruned Execution Time, Average Power, Total Energy & Energy Efficiency.

Avg CPU (%)	Avg RAM (%)
99.442	69.818

Figure 21: Average CPU & RAM Usage tinylama pruned.

llama3.2 - pruned

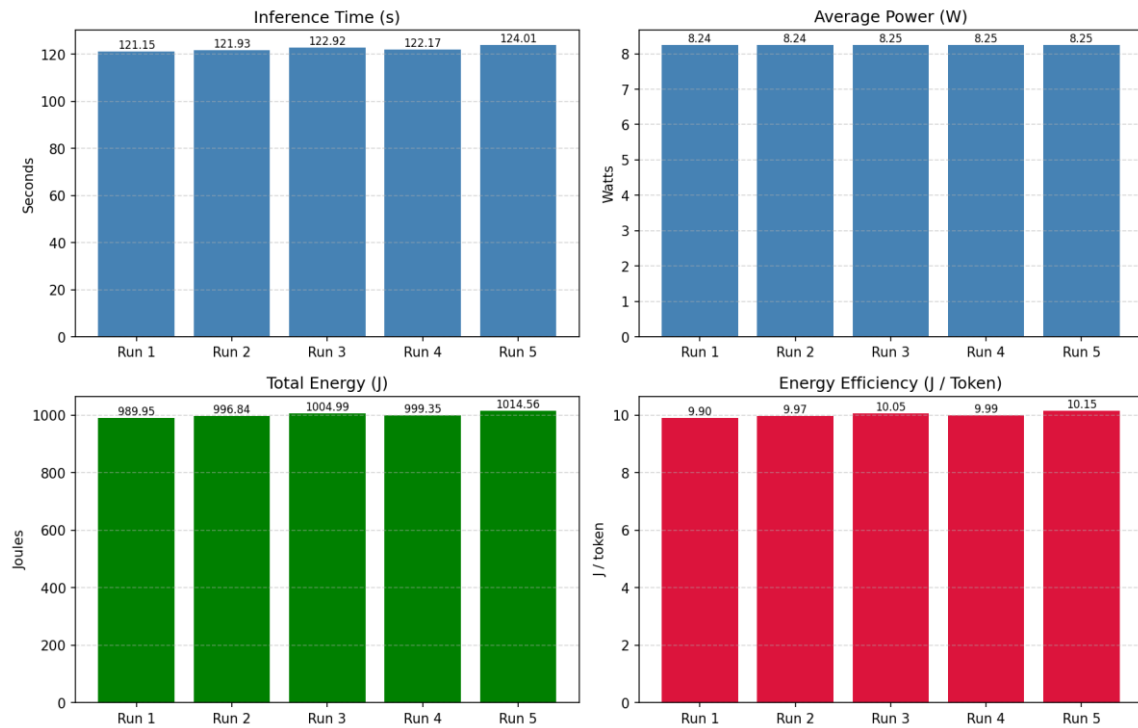


Figure 22: llama3.2 pruned Execution Time, Average Power, Total Energy & Energy Efficiency.

Avg CPU (%)	Avg RAM (%)
99.704	77.212

Figure 23: Average CPU & RAM Usage llama3.2 pruned.

qwen2 - pruned

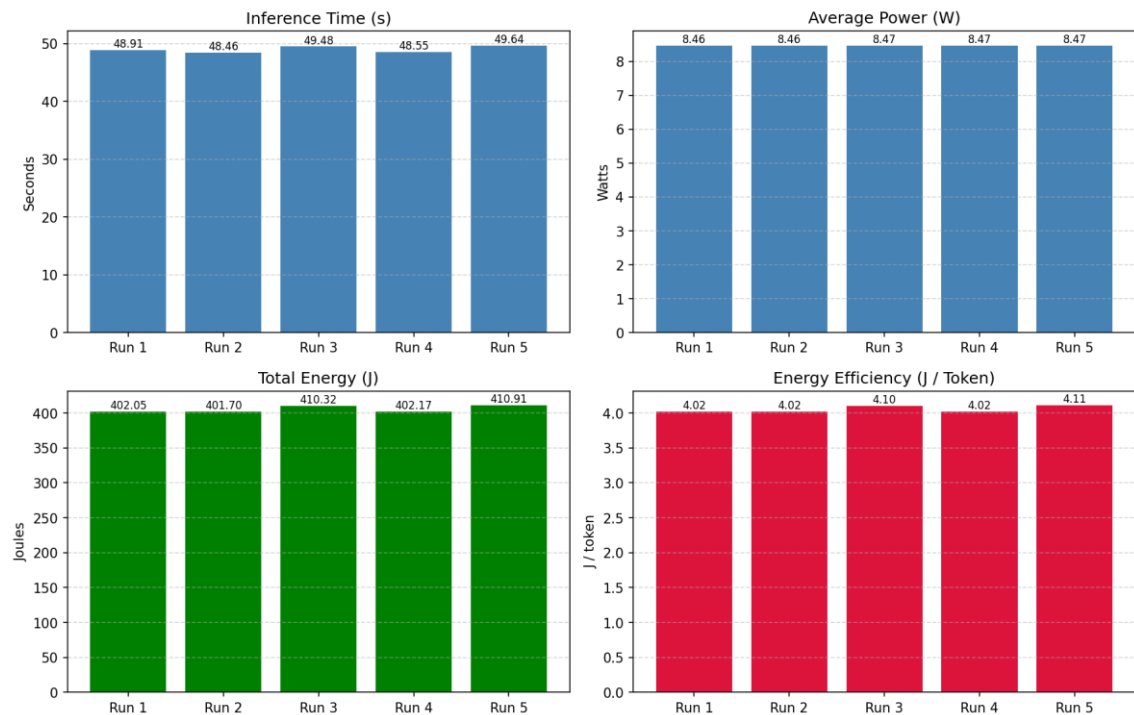


Figure 24: qwen2 pruned Execution Time, Average Power, Total Energy & Energy Efficiency.

Avg CPU (%)	Avg RAM (%)
99.166	37.548

Figure 25: Average CPU & RAM Usage qwen2 pruned.

In addition, degraded response quality was observed for Llama3.2/pruned and Qwen2/pruned: repeated tokens and phrases. This suggests that 30% pruning without retraining significantly harms these models, whereas TinyLlama/pruned remained coherent, but with content errors.

5.3.4 DPS Method

The DPS method exhibits the most interesting behaviour of the experiment from an academic point of view. The three models produced three different switching patterns, which reflect the relationship between model size and the computational capability of the Pi 400.

Table 5: DPS behaviour per model. Thresholds: slow=0.80 s/token, fast=0.65 s/token.

Model	Switches	Pattern	J/token	Remark
TinyLlama	4	fp16→q8→fp16→q8→fp16	7.316	Oscillation, borderline performance
Llama3.2	1	fp16→q8→q8→q8→q8	6.391	One switch, steady q8
Qwen2	0	fp16×5	4.104	No switch, fast fp16

- TinyLlama (1.1B) exhibits an oscillation of 4 switches: fp16→q8→fp16→q8→fp16. The reason is that fp16 (~1.09–1.11 s/token) always lies above the slow threshold (0.80), while q8 (~0.59–0.60 s/token) always lies below the fast threshold (0.65). Since both precisions lie outside the hysteresis band [0.65, 0.80], the switching logic inevitably leads to a precision change in every run.

This behaviour shows that the hysteresis gap on its own does not prevent oscillation when the performance of the two precisions lies on either side of the band. The cause is that the return to fp16 is decided on the basis of the speed of Q8_0, which does not predict the speed of fp16. A real application would require a moving average of s/token, a separate s/token history per precision, or a minimum number of runs at each precision.

- Llama3.2 (1.2B) runs at ~1.21 s/token, a value above the slow threshold. It switches to q8 at run 2 and stays there for the remaining 4 runs (q8: ~0.652–0.665 s/token, just above the fast threshold of 0.65).

Result: 1 switch, a 35.7% reduction in energy per token compared with fp16.

- Qwen2 (0.5B parameters) runs at ~ 0.50 s/token, a value below the slow threshold (0.80), so the switch to Q8_0 is never triggered.

Result: no switch, full fp16 precision in all runs and energy per token equal to fp16 (+0.2%, within measurement variation). This shows that, when fp16 is fast enough, DPS maintains full precision at no additional energy cost.

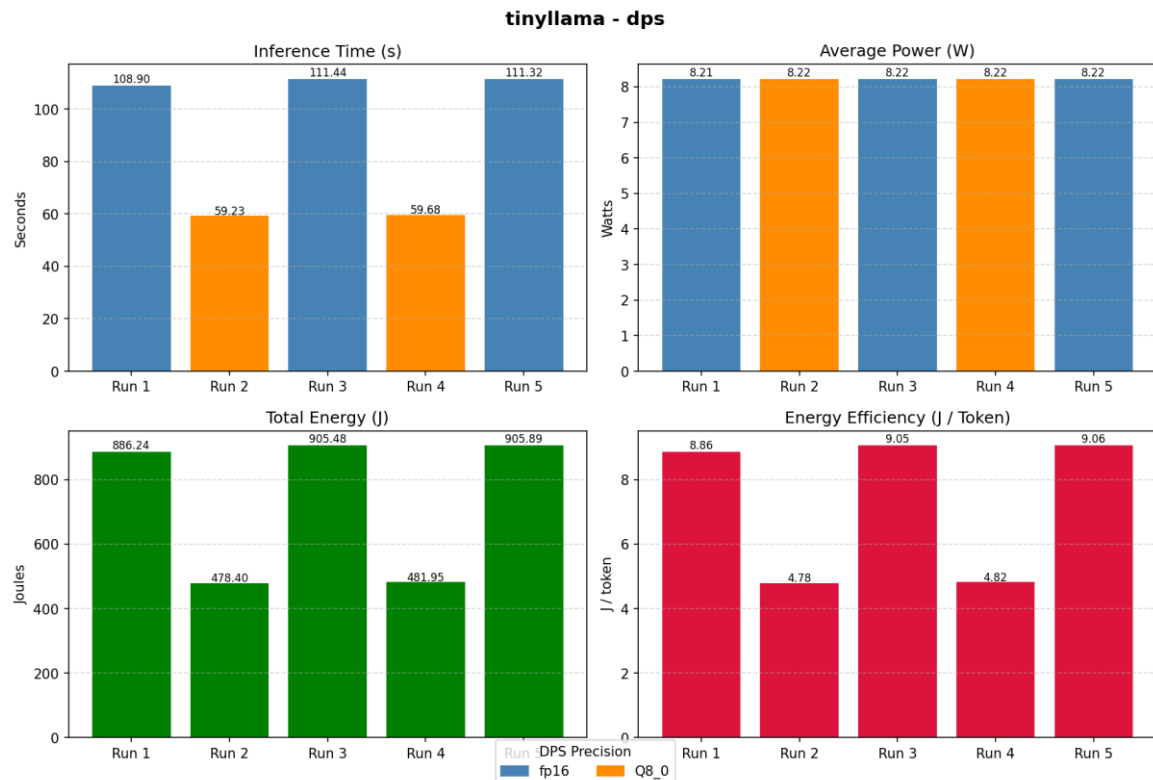


Figure 26: tinylama dps Execution Time, Average Power, Total Energy & Energy Efficiency.

Avg CPU (%)	Avg RAM (%)
99.778	92.238

Figure 27: Average CPU & RAM Usage tinylama dps.

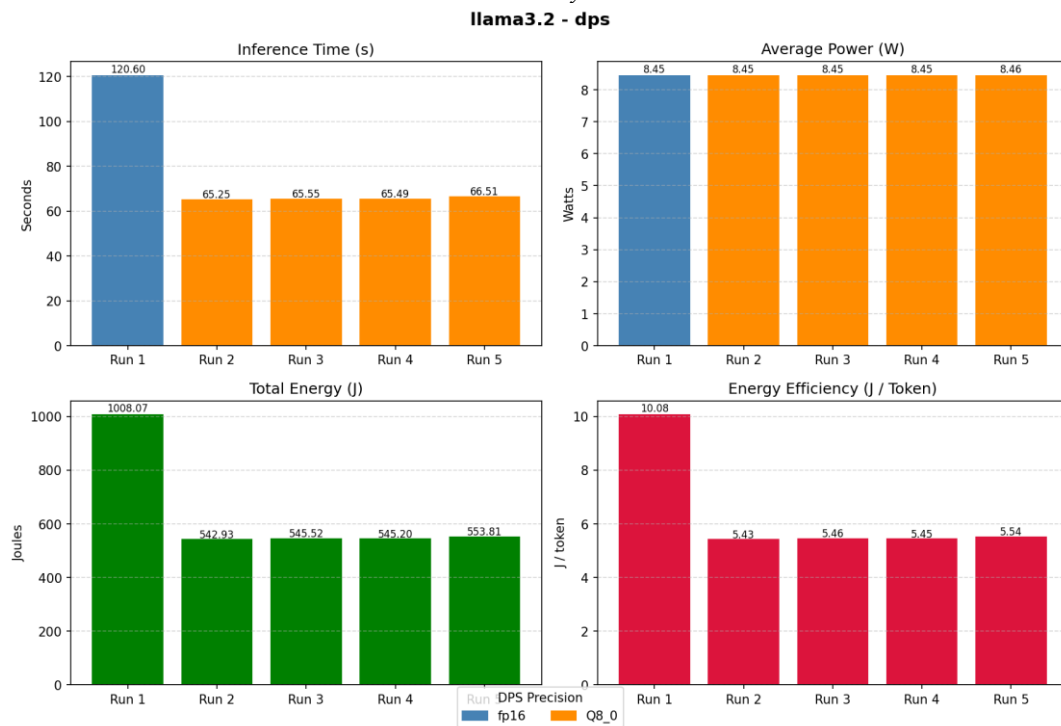


Figure 28: llama3.2 dps Execution Time, Average Power, Total Energy & Energy Efficiency.

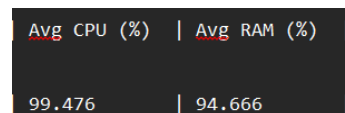


Figure 29: Average CPU & RAM Usage llama3.2 dps.

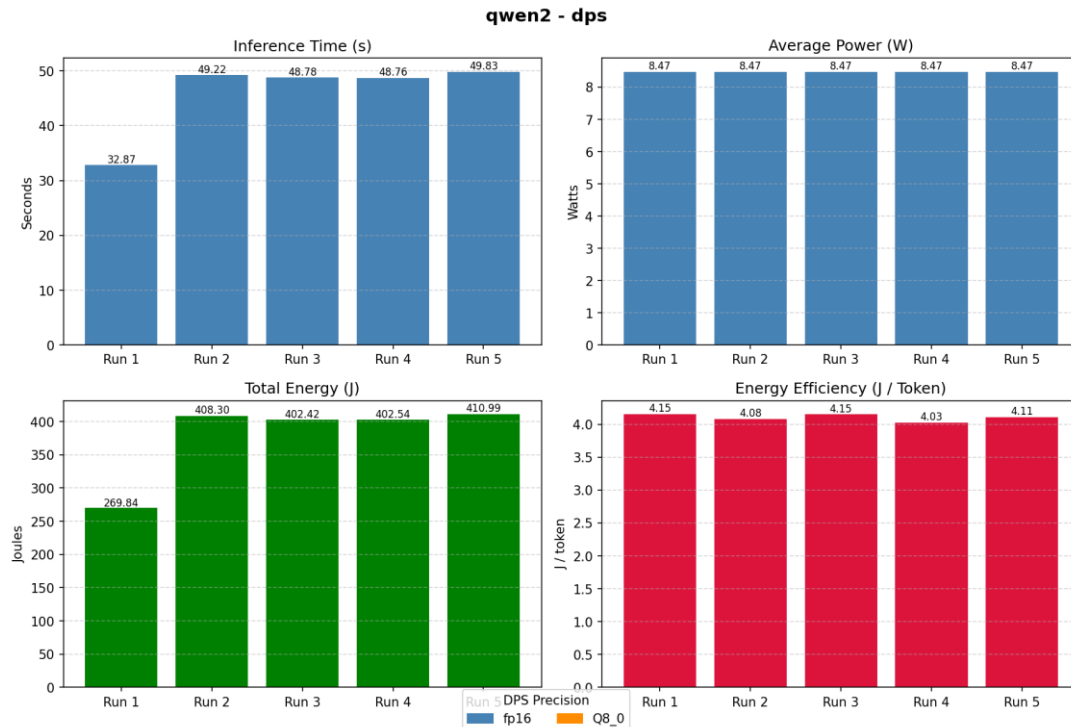


Figure 30: qwen2 dps Execution Time, Average Power, Total Energy & Energy Efficiency.

Avg CPU (%)	Avg RAM (%)
99.144	37.512

Figure 31: Average CPU & RAM Usage qwen2 dps.

5.4 Comparative Analysis of Methods

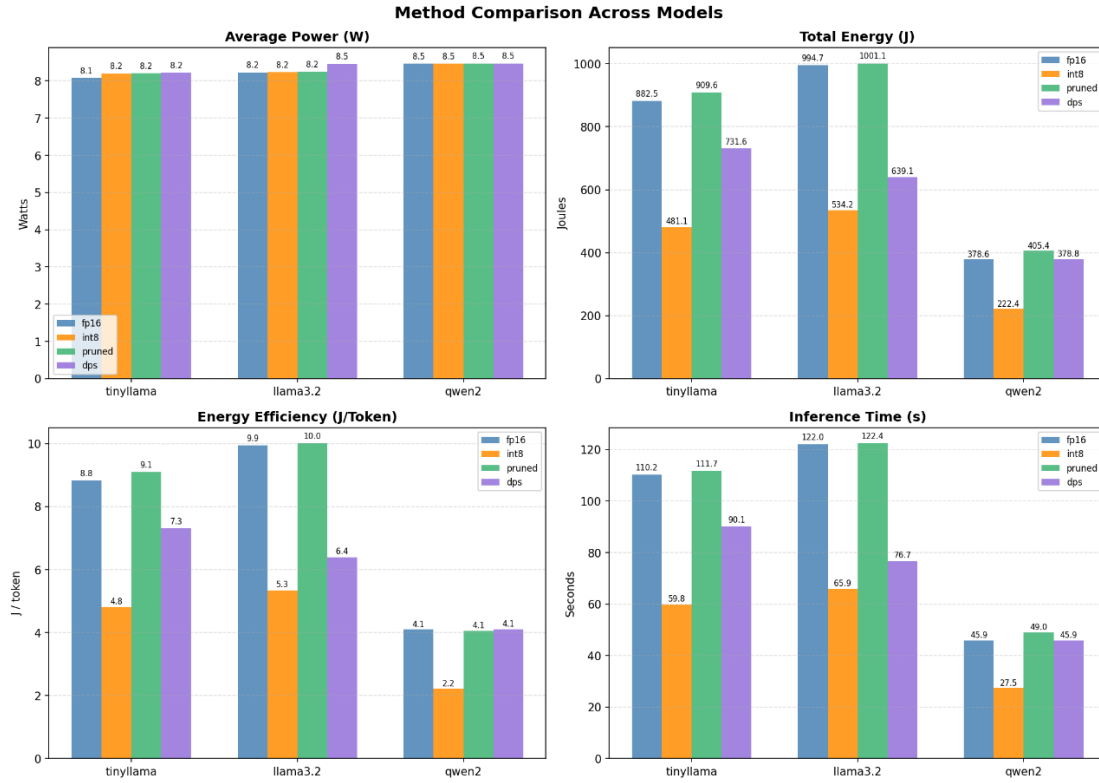


Figure 32: Comparison of Methods Across All Models

Table 6 summarises the relative change in energy (J/token) of each method compared with the fp16 baseline. A negative value means an improvement (less energy), a positive value a deterioration.

Table 6: Relative change in J/token versus the fp16 baseline.

Model	int8 vs fp16	pruned vs fp16	dps vs fp16	dps vs int8	Best
TinyLlama	-45.5%	+3.1%	-17.1%	+52.1%	int8
Llama3.2	-46.3%	+0.7%	-35.7%	+19.6%	int8
Qwen2	-45.7%	-1.0%	+0.2%	+84.5%	int8

The comparative analysis leads to the following observations:

- **INT8 Quantization:** Provides the most consistent and largest energy reduction (~45%) across all models, with no apparent quality degradation for TinyLlama and Llama 3.2. It is the recommended method for CPU inference on constrained hardware.

- **Unstructured Pruning:** Yields no energy benefit in this setup. To exploit sparsity for energy savings, structured pruning (removal of attention heads or layers) or hardware with native sparse execution support is required.
- **DPS:** A runtime-adaptive strategy of considerable interest. On the Pi 400 it does not outperform int8 in energy terms (because it executes some of the runs in fp16, while model loading time is not included in the measurements), but its value lies in its adaptability: on faster hardware it would keep fp16 for small models and switch to int8 only for larger ones.

5.5 Effect of Model Size

An important finding of the experiment is that model size affects energy consumption more than the choice of method. Qwen2 fp16 (4.096 J/token) is more efficient than TinyLlama int8 (4.811 J/token) and Llama3.2 int8 (5.342 J/token); that is, a model at full precision without any optimisation beats another, quantised model. This shows that choosing the right model from the outset can be more important than any optimisation method.

This suggests that for applications in an embedded environment where the main goal is energy efficiency, choosing a smaller model is more effective than any optimisation method. The optimal choice remains the combination: small model + int8 quantization.

5.6 Final Conclusions

The experimental results lead to the following conclusions:

1. The power of the Raspberry Pi 400 remains almost constant (8.1 to 8.5 W) regardless of inference method. The main factor that determines energy consumption is execution time.
2. INT8 quantization consistently delivers ~45% energy savings across all models, with no apparent quality cost in the two larger models. It is the recommended method for energy-efficient inference on CPUs.
3. Unstructured pruning brings no energy benefit in CPU inference with llama.cpp. Structured pruning or specialised sparse hardware is required to exploit sparsity.
4. DPS demonstrates the functionality of the adaptive mechanism: different models exhibit different behaviour (0, 1 or 4 switches) depending on how their performance relates to the defined thresholds.
5. Model size has a greater effect on energy than any optimisation method. The optimal combination for embedded systems concerned only with energy efficiency is: choosing a smaller model + INT8.

Chapter 6: Conclusions and Future Work

This chapter summarises the conclusions of the thesis, analyses the problems encountered during implementation and how they were resolved, critically evaluates the overall approach, and presents proposals for future research and extension of the work.

6.1 Overall Conclusions of the Thesis

This thesis aimed to measure and compare the energy consumption of four inference methods for language models (LLMs) in a resource-constrained environment, specifically on the Raspberry Pi 400. The methods examined were full-precision execution (fp16), INT8 quantization (Q8_0/INT8), weight sparsification (unstructured pruning 30%), and dynamic precision control (Dynamic Precision Scaling/DPS).

The central finding of the thesis is that power consumption remains almost constant (8.1 to 8.5 W) regardless of method, so that energy efficiency depends mainly on inference time. INT8 quantization achieved a ~45% reduction in energy consumption per token across all models, without a noticeable degradation of response quality for TinyLlama and Llama 3.2. Unstructured pruning yielded no energy benefit due to the architectural limitations of CPU inference. DPS demonstrated the operation of an adaptive mechanism that responds dynamically to hardware conditions.

An equally important finding is that the choice of model affects energy consumption more than any optimisation method: Qwen2 fp16 (4.096 J/token) proved more efficient than TinyLlama int8 (4.811 J/token), showing that combining a small model with INT8 is the optimal strategy for energy-efficient embedded environments.

6.2 Problems Encountered

During the implementation of the thesis, several technical problems were encountered that required parts of the architecture to be redesigned. The most important are presented below.

6.2.1 Redesign of the DPS Logic

The initial implementation of the DPS method was based on power thresholds: if the measured power exceeded 6.0 W, the system switched from fp16 to Q8_0, while below 5.0 W it returned to fp16 (with CPU and RAM usage as secondary criteria). The problem became apparent as soon as measurements began: the Raspberry Pi 400 consistently draws ~7.5 W at idle, so the threshold was always exceeded from the first run, meaning that DPS was in effect no different from plain INT8.

The solution was to redesign the switching logic around inference time (seconds-per-token). Since power is constant, s/token is a direct proxy for energy efficiency.

Two thresholds with hysteresis were defined (slow: 0.80 s/token, fast: 0.65 s/token), calibrated from actual measurements on the Pi 400. This change made the DPS logic well founded and measurable.

6.2.2 Decoding the UD18 Protocol

The Atorch UD18 sensor has no official documentation of its binary communication protocol. The data are sent as raw bytes over Bluetooth RFCOMM, and decoding them requires knowledge of the exact byte offsets for each measurement.

The problem was solved through a combination of reverse engineering and test measurements: the values shown on the UD18 display were compared with the values produced by the parsing code for various known loads. In this way the offsets for voltage (bytes 4–6, uint24/100), current (bytes 7–9, uint24/100) and power (bytes 22–25, uint32/1000) were verified. The checksum byte is not verified in the final implementation. This process, although time-consuming, led to a reliable and verifiable implementation.

6.2.3 Memory Management on the Raspberry Pi

The Raspberry Pi 400 has 4 GB of RAM, an amount that is borderline for loading and running LLMs in fp16. During the first tests, running successive methods in the same Python process led to gradual memory exhaustion, resulting in increasing inference times due to swapping and non-reproducible results.

The solution was twofold: first, each model/method combination is executed as an independent subprocess, ensuring full release of memory between methods. Second, `experiment_runner.py` writes '3' to `/proc/sys/vm/drop_caches` before and after each subprocess, clearing the operating system's page cache. In addition, swap was increased to 2 GB to cope with peak cases. This approach significantly improved the reproducibility of results between runs.

6.3 Critical Evaluation

Beyond the experimental results, it is important to evaluate critically both the strengths of the implementation and the limitations that affect the generalisation of the findings.

6.3.1 Strengths of the Implementation

- **Reproducibility.** The combination of subprocess isolation, `drop_caches` and a fixed temperature (0.1) ensures high reproducibility. As an indication, the DPS method on Qwen2, which never changed precision, was in practice an independent repetition of fp16 and gave almost the same energy (4.104 versus 4.096 J/token, a deviation of 0.2%) and the same responses.
- **Modular architecture.** The separation into Inference, Monitoring, Evaluation and Experiment modules allows easy extension: new models can be added mainly through `config.yaml`.
- **Hardware-level measurement.** Using an external sensor (Atorch UD18) instead of software-based estimation provides real power measurements independent of the software being measured, thereby limiting the observer effect.
- **Well-founded DPS logic.** Time-based DPS with hysteresis is theoretically well founded and academically defensible: the switching decision is based on measured data, not on estimates.

6.3.2 Limitations and Weaknesses

- **A single hardware environment.** All experiments were run exclusively on the Raspberry Pi 400. The results, particularly for DPS, cannot be directly generalised to other hardware. On faster processors the thresholds would need to be recalibrated.

- Small number of models. Only three models in the 0.5 to 1.2B parameter range were examined. Extending the study to larger models (3B, 7B) or different architectures (Mistral, Gemma) would strengthen the generalisability of the conclusions.
- Unstructured pruning without retraining. The 30% L1 unstructured pruning was applied without retraining (fine-tuning). In the literature, pruning without retraining rarely yields an energy benefit in CPU inference, so the negative result was expected and confirms the theory, but it does not represent the best possible pruning implementation.
- Fixed workload. Each run generates up to 100 tokens (`max_new_tokens = 100`). Most runs reach the limit, with the exception of some Qwen2 runs (fp16 and dps, average 92.4 tokens) that terminated earlier. In real applications the length of responses varies considerably, which affects the behaviour of DPS and the J/token measurements.
- Statistical and evaluation limitations. Averages of 5 runs are presented without standard deviation, the order of execution was not randomised (average power rises gradually from 8.08 to 8.47 W with the order of execution) and response quality was assessed only qualitatively. Reporting the standard deviation, randomising the order of execution and using metrics such as perplexity would strengthen the conclusions.

6.4 Future Work and Extensions

The findings of this thesis open up multiple directions for future research. The most important proposals for extension are presented below.

6.4.1 Application to Larger LLMs

This thesis focused on models in the 0.5B to 1.2B parameter range, which is the category that can be run on the Raspberry Pi 400. Future work could extend the methodology to models of 3B–7B parameters, using more powerful hardware (Raspberry Pi 5, NVIDIA Jetson Nano, Apple Silicon)[5].

Of particular interest is whether the constant power observed on the Pi 400 is a characteristic of this category of hardware, or whether on other platforms power varies significantly between methods, which would radically change the conclusions about the optimal method.

6.4.2 DPS on Faster Hardware

DPS demonstrated the operation of the adaptive mechanism on the Pi 400, but this hardware is borderline slow for fp16 inference with 1B+ models. As a result, for the ~1B models DPS switches to Q8_0 or oscillates, limiting the academic value of its adaptability.

On faster hardware (e.g. a Raspberry Pi 5 with 8 GB of RAM or Apple M-series), the fp16 baseline for models such as TinyLlama and Llama 3.2 would lie below the slow threshold, and DPS would maintain full precision for small to medium models and switch to Q8_0 only for larger ones. This behaviour would fully demonstrate the claim that DPS works as a strategy that adapts to the hardware without manual configuration.

6.4.3 Research on Thermally-Aware DPS AI Inference

During the experiment it was observed that inference time increased gradually after many successive runs and that average power increased with the order of execution (from 8.08 to 8.47 W). A possible explanation is the rise in processor temperature and possibly thermal throttling,

since the Raspberry Pi 400 has no active cooling. However, temperature was not recorded in this experiment, so this hypothesis requires confirmation.

Future work could examine the development of Thermally-Aware inference [22]: a system that monitors CPU temperature in real time (through `/sys/class/thermal`) and uses it as an additional signal for precision switching decisions, in other words combining the DPS logic with thermal management.

References

- [1] A. d’Orgeval, S. Sheehan, Q. Avenas, E. Assoumou, and V. Sessa, ‘Generative AI impact assessment through a life cycle analysis of multiple data center typologies’, *Appl. Energy*, vol. 406, p. 127288, Mar. 2026, doi: 10.1016/j.apenergy.2025.127288.
- [2] A. Alaktif, M. Chergui, I. Daoudi, and A. Ammoumou, ‘All You Should Know About Large Language Models (LLMs)’, in *2024 7th International Conference on Advanced Communication Technologies and Networking (CommNet)*, Dec. 2024, pp. 1–10. doi: 10.1109/CommNet63022.2024.10793270.
- [3] X. Zhang *et al.*, ‘Edge Intelligence in the Generative Artificial Intelligence Era’, *IEEE Wirel. Commun.*, vol. 32, no. 6, pp. 60–68, Dec. 2025, doi: 10.1109/MWC.2025.3599652.
- [4] A. Dequino, L. Bompani, L. Benini, and F. Conti, ‘Optimizing BFloat16 Deployment of Tiny Transformers on Ultra-Low Power Extreme Edge SoCs’, *J. Low Power Electron. Appl.*, vol. 15, no. 1, p. 8, Mar. 2025, doi: 10.3390/jlpea15010008.
- [5] M. R. Islam *et al.*, ‘Characterizing and Understanding Energy Footprint and Efficiency of Small Language Model on Edges’, in *2025 IEEE 22nd International Conference on Mobile Ad-Hoc and Smart Systems (MASS)*, Oct. 2025, pp. 363–371. doi: 10.1109/MASS66014.2025.00059.
- [6] H. Martínez, S. Catalán, A. Castelló, and E. S. Quintana-Ortí, ‘Characterization of quantized inference with transformer encoders on low power CPUs’, *Int. J. High Perform. Comput. Appl.*, vol. 39, no. 6, pp. 803–821, Nov. 2025, doi: 10.1177/10943420251355115.
- [7] S. O. Semerikov, T. A. Vakaliuk, O. B. Kanevska, O. A. Ostroushko, and A. O. Kolhatin, ‘Edge intelligence unleashed: a survey on deploying large language models in resource-constrained environments’, *J. Edge Comput.*, vol. 4, no. 2, pp. 179–233, Nov. 2025, doi: 10.55056/jec.1000.
- [8] S. S. Girija, L. Arora, S. Kapoor, D. Pradhan, A. Raj, and A. Shetgaonkar, ‘Optimizing LLMs for Resource-Constrained Environments: A Survey of Model Compression Techniques’, in *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*, Jul. 2025, pp. 1657–1664. doi: 10.1109/COMPSAC65507.2025.00224.
- [9] C. Tian *et al.*, ‘A survey on deep learning fundamentals’, *Artif. Intell. Rev.*, vol. 58, no. 12, p. 381, Oct. 2025, doi: 10.1007/s10462-025-11368-7.
- [10] A. Vaswani *et al.*, ‘Attention Is All You Need’, Aug. 02, 2023, *arXiv*: arXiv:1706.03762. doi: 10.48550/arXiv.1706.03762.
- [11] S. Sun *et al.*, ‘Optimizing Attention for Efficient LLM Inference: A Review’, in *2025 8th World Conference on Computing and Communication Technologies (WCCCT)*, Apr. 2025, pp. 482–491. doi: 10.1109/WCCCT65447.2025.11027973.
- [12] I. Mangkunegara, I. Purwono, A. Ma’arif, D.-N. Basil, H. M. Marhoon, and A.-N. Sharkawy, ‘Transformer Models in Deep Learning: Foundations, Advances, Challenges and Future Directions’, *Bul. Ilm. Sarj. Tek. Elektro*, vol. 7, pp. 231–241, Jun. 2025, doi: 10.12928/biste.v7i2.13053.
- [13] P. V. Dantas, L. C. Cordeiro, and W. S. S. Junior, ‘A review of state-of-the-art techniques for large language model compression’, *Complex Intell. Syst.*, vol. 11, no. 9, p. 407, Aug. 2025, doi: 10.1007/s40747-025-02019-z.
- [14] J. Lang, Z. Guo, and S. Huang, ‘A Comprehensive Study on Quantization Techniques for Large Language Models’, in *2024 4th International Conference on Artificial Intelligence*,

Robotics, and Communication (ICAIRC), Dec. 2024, pp. 224–231. doi: 10.1109/ICAIRC64177.2024.10899941.

[15] E. J. Husom *et al.*, ‘Sustainable LLM Inference for Edge AI: Evaluating Quantized LLMs for Energy Efficiency, Output Accuracy, and Inference Latency’, *ACM Trans Internet Things*, vol. 6, no. 4, p. 28:1–28:35, Nov. 2025, doi: 10.1145/3767742.

[16] S. Hashemi, N. Anthony, H. Tann, R. I. Bahar, and S. Reda, ‘Understanding the impact of precision quantization on the accuracy and energy of neural networks’, in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, Mar. 2017, pp. 1474–1479. doi: 10.23919/DATE.2017.7927224.

[17] H. Cheng, M. Zhang, and J. Q. Shi, ‘A Survey on Deep Neural Network Pruning: Taxonomy, Comparison, Analysis, and Recommendations’, *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 46, no. 12, pp. 10558–10578, Dec. 2024, doi: 10.1109/TPAMI.2024.3447085.

[18] C. Tian, X. Qin, and L. Li, ‘GreenLLM: Towards Efficient Large Language Model via Energy-aware Pruning’, in *2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS)*, Jun. 2024, pp. 1–2. doi: 10.1109/IWQoS61813.2024.10682928.

[19] L. Liu *et al.*, ‘Drift: Leveraging Distribution-based Dynamic Precision Quantization for Efficient Deep Neural Network Acceleration’, in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, in DAC ’24. New York, NY, USA: Association for Computing Machinery, Nov. 2024, pp. 1–6. doi: 10.1145/3649329.3655986.

[20] M. R. P C, A. M R, V. P. Gopi, G. Lakshminarayanan, G. R. Gangadharan, and J. U. Kidav, ‘Dynamic Precision Scaling in MAC Units for Energy-Efficient Computations in Deep Neural Network Accelerators’, in *2024 28th International Symposium on VLSI Design and Test (VDATE)*, Sep. 2024, pp. 1–7. doi: 10.1109/VDATE63601.2024.10705697.

[21] J. Stojkovic, C. Zhang, Í. Goiri, J. Torrellas, and E. Choukse, ‘DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency’, in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Mar. 2025. doi: 10.1109/HPCA61900.2025.00102.

[22] J. Stojkovic *et al.*, ‘TAPAS: Thermal- and Power-Aware Scheduling for LLM Inference in Cloud Platforms’, in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’25)*, Mar. 2025. doi: 10.1145/3676641.3716025.

[23] Z. Huang *et al.*, ‘MNN-AECS: Energy Optimization for LLM Decoding on Mobile Devices via Adaptive Core Selection’, Jun. 2025, *arXiv*: arXiv:2506.19884. doi: 10.48550/arXiv.2506.19884.

[24] Q. Liu, D. Huang, M. Zapater, and D. Atienza, ‘GreenLLM: SLO-Aware Dynamic Frequency Scaling for Energy-Efficient LLM Serving’, Aug. 2025, *arXiv*: arXiv:2508.16449. doi: 10.48550/arXiv.2508.16449.

Appendix 1. Developed Code

The code developed during the preparation of this thesis, with full explanation, can be found in the following GitHub repository:

https://github.com/TsipiDev/Thesis_Code_Dimitris_Vatousis_220007