SPRINT: Sensitivity-guided Pruning for Inference-Time Adaptation of LLMs

by

Asief Iqbal Dieyaz
23341081
Nahid Hassan
22101822
Faiyaz Bin Yousuf
22101845

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science and Engineering

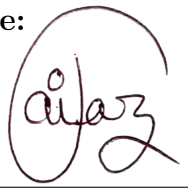
Department of Computer Science and Engineering
Brac University
April 2026

Declaration

It is hereby declared that

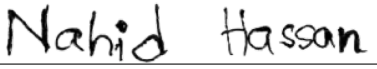
1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:



Faiyaz Bin Yousuf
22101845

Asief Iqbal Dieyaz
23341081



Nahid Hassan
22101822

Approval

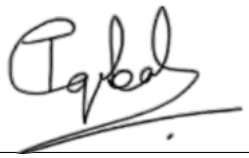
The thesis/project titled "SPRINT: Sensitivity-guided Pruning for Inference-Time Adaptation of LLMs" submitted by

1. Nahid Hassan (22101822)
2. Asief Iqbal Dieyaz (23341081)
3. Faiyaz Bin Yousuf (22101845)

of Spring, 2026 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science in April 12,2026.

Examining Committee:

Supervisor:
(Member)



Muhammad Iqbal Hossain, PhD

Associate Professor
Department of Computer Science and Engineering
Brac University

Thesis Coordinator:
(Member)

Dr. Md. Golam Rabiul Alam

Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Sadia Hamid Kazi, PhD

Chairperson and Associate Professor
Department of Computer Science and Engineering
Brac University

Abstract

Large Language Models have transformed artificial intelligence, showing strong performance across a wide range of natural language processing tasks; however, their computational and memory demands make deployment on resource-constrained hardware, such as personal laptops, local servers, and edge devices, largely impractical. Existing compression techniques including pruning, quantization, knowledge distillation, and Low-Rank Adaptation reduce model overhead but apply a fixed compression level at deployment time, leaving them unable to respond to changes in prompt complexity, available memory, or latency requirements during inference. This work presents SPRINT, a dynamic hybrid inference framework that selects pruning intensity on a per-prompt basis at runtime. The system consists of three components: a Learned Complexity Router (LCR) built on a fine-tuned BERT-mini backbone that predicts each prompt's sensitivity to pruning, a Double Deep Q-Network (DDQN) controller that selects pruning actions from a 17-option discrete space using a 10-dimensional state vector combining hardware telemetry, router scores, and early backbone signals, and a structural pruning engine that physically removes transformer layers to produce real latency reductions. Experiments were conducted on Llama-2-7B using a 10,000-prompt dataset drawn equally from GSM8K, MBPP, WikiText-2, MMLU, and BoolQ. The LCR achieved a Spearman rank correlation of $\rho = 0.797$ (95% CI: [0.779, 0.817]) and $R^2 = 0.633$ against oracle sensitivity labels on the held-out test set, exceeding the target threshold of $\rho \geq 0.70$ across all five benchmark domains. Across 2,000 held-out test episodes, the DDQN controller reduced average inference time from 1,287.61 ms to 875.39 ms, yielding a 32.0% average speedup. Parameter count dropped from 4,714.3 MB to 3,113.5 MB, a reduction of 1,600.8 MB (34.0%), while peak VRAM consumption remained stable at approximately 4.75 GB. Total routing and action-selection overhead averaged just 17.77 ms, corresponding to approximately 2.0% of total latency. Comparison against SparseGPT, Wanda, and LLM Pruner confirmed that unstructured weight sparsity does not reliably convert to latency reduction on standard GPU hardware, whereas SPRINT's structural layer removal yields predictable speedups without requiring sparse kernel support.

Keywords: Large Language Models, Dynamic Optimization, Adaptive Inference, Structural Pruning, Reinforcement Learning, Resource-Aware Systems, Edge Deployment, DDQN

Acknowledgement

First and foremost, all praise and gratitude are due to the Almighty Allah for providing us with the strength, knowledge, and perseverance to complete our Thesis paper successfully. We would also like to express our gratitude to our supervisor, Muhammad Iqbal Hossain, sir, for giving us the best level of guidelines and support for our successful completion. His constant guidance and encouragement have been the perfect motivation for us to refine our ideas and put ample effort into the paper. Lastly, we are very much thankful to our parents, families, colleagues, and friends who have constantly supported us in fulfilling our goals. May Allah bless them all.

Table of Contents

List of Figures

Chapter 1
Introduction

1.1 Background

In recent years, Large Language Models (LLMs) have made significant advancements in the field of artificial intelligence. Open-source LLMs such as BERT, Stable Diffusion, Llama, etc have created opportunities, including high-end text analytics and code efficiency, to the design of digital art. Such models are constructed on elaborate designs of billions of parameters, which enable them to comprehend and impressively produce human-like content with remarkable accuracy. The issue is that such LLMs need immense amounts of computational power and intensive resources. They typically require advanced processing tools, storage, etc., and high-end memory, which is not affordable to most people with standard devices. When a person wishes to run an LLM on a weaker machine, they must manually tweak its settings. This involves a lot of trial and error, adjusting things like model size through pruning or reducing the precision of its calculations through quantization. This process is time-consuming and requires a lot of technical expertise. Due to such difficulties, a widening disparity exists between the capabilities of these LLMs and the people capable of utilizing them. Consequently, this leads to a barrier to accessibility, preventing a wider range of users from benefiting from advanced AI capabilities.

1.2 Rationale of the Study or Motivation

The main motivation behind this research is to break down the hardware barriers that currently limit the use of LLMs. Right now, the way we deploy these models is largely static; a model is released with a single configuration that assumes the user has powerful hardware. This "one-size-fits-all" approach is inefficient and exclusive to specific hardware and resources. It's similar to a modern video game being released with only "Ultra" graphics settings, making it unplayable on most standard computers. In modern video games, developers solve this by offering a range of graphics settings (low, medium, high) and often an "auto-detect" feature that configures the game according to the device the game is being played. A similar paradigm is necessary for LLM deployment where based on the available computational resources, an LLM's behaviour can be adapted dynamically.
This study proposes to develop a resource-aware adaptive system for the inference of large language models. The goal is to create a compression framework that can first analyze the user's hardware—identifying the type of processing units, the amount of available memory, storage, and model behaviour, and also incorporates runtime signals to guide pruning decisions dynamically. This then helps automatically adjust the LLM's parameters for the best possible performance on that device. It's not just about making the model run; it's about making it run optimally. This means finding the perfect balance between speed and accuracy for that specific hardware, ensuring the user gets the best experience their device can offer.

By creating such an adaptive inference framework, it can democratize the use of LLMs across a wide range of users. Students could run sophisticated models on their personal laptops for research, developers in smaller companies could integrate AI features into their mobile apps without worrying about performance issues, and innovators in resource-constrained environments could experiment with LLMs more efficiently. As LLMs are becoming part of our daily lives,there is a growing need for systems that are powerful and also adaptable and efficient. They need to be more inclusive and efficient, moving away from a static deployment model to a dynamic one where powerful tools adapt to the user, not the other way around.

1.3 Problem Statement

Running large language models on consumer hardware remains a fundamental resource allocation challenge. Models such as Llama-2-7B require substantial VRAM in half-precision, significant CPU memory for KV-cache accumulation, and sustained memory-bandwidth utilization that quickly exceeds the practical limits of edge hardware—personal computers, local servers, and single-GPU workstations. Static compression techniques have become the dominant practical response to this constraint. Quantization, magnitude-based pruning, and knowledge distillation each reduce the model footprint permanently before deployment and leave that configuration fixed for all subsequent inference.

This creates a fundamental mismatch between the assumptions underlying static compression and the actual requirements of local inference. A fixed pruning mask cannot distinguish between computationally simple prompts and complex multi-step reasoning problems. Both receive the same structural configuration, even though simple prompts are empirically far more tolerant of layer removal and attention-head pruning. A compression profile calibrated for one system state cannot adapt when available GPU memory fluctuates because another application has consumed resources, or when thermal throttling has reduced effective computational capacity mid-session. The model has no mechanism to account for these variations.

Hybrid compression schemes that stack multiple static operators—pruning followed by quantization, or weight sparsification combined with distillation—address the footprint problem more aggressively but inherit the same structural inflexibility. The compression decision is made once at export time, and the deployment environment is assumed static thereafter. This assumption does not reflect what local inference looks like in practice.

The specific gap this research addresses is the absence of a per-prompt, per-state pruning controller that integrates three information sources: an estimate of how sensitive the current prompt is to structural compression, the live state of system resources at the moment of inference, and early internal signals from the backbone model itself. Existing runtime pruning approaches use heuristic complexity proxies such as token count and surface perplexity thresholds. These do not directly measure how the target backbone degrades under the specific pruning operators being deployed. They do not produce operator-specific sensitivity estimates, and they do not adapt to hardware telemetry at the granularity of individual inference calls. No published framework combines a learned oracle-derived sensitivity router, operator-specific labels, and a reinforcement learning controller with live hardware state into a single adaptive inference pipeline.

The practical result is that current local inference systems must choose between two inadequate options: run the full model and accept the memory and latency cost, or

apply a fixed aggressive compression and accept the quality degradation on every prompt, including those that can tolerate structural sparsification without meaningful performance loss.

1.4 Objective

This thesis designs, implements, and evaluates SPRINT (Sensitivity-guided Pruning for Inference-Time Adaptation), a framework that makes per-prompt structural pruning decisions at inference time using a learned complexity router, live hardware telemetry, and a reinforcement learning controller. The goal is to demonstrate that structural pruning intensity can be selected dynamically, per prompt, without relying on fixed offline compression profiles, and that the resulting system achieves measurable inference speedup while bounding quality degradation within acceptable limits.
The research objectives are as follows.

1. Construct a multi-domain oracle sensitivity dataset by evaluating a backbone LLM under dense and sparse configurations across a diverse benchmark mixture drawn from multiple public sources, with balanced representation across mathematical reasoning, code generation, narrative language modeling, multiple-choice reasoning, and question answering. Produce per-prompt, per-operator sensitivity labels that directly measure backbone degradation under attention-head pruning and transformer-layer removal, rather than relying on surface heuristics.

2. Train a Learned Complexity Router based on a lightweight transformer model (BERT-mini) extended with multi-layer feature aggregation, attention statistics extraction, and auxiliary text-level features, to predict prompt pruning sensitivity at inference time. Achieve high correlation between predictions and oracle sensitivities on held-out test data, demonstrating that the router generalizes across prompt types and domains.

3. Design and train a reinforcement learning controller using a Double Deep Q-Network architecture that integrates a multi-dimensional state vector comprising hardware telemetry, the learned sensitivity score, and early backbone signals to select from a discrete action space of pruning configurations spanning both layer-skipping and attention-head pruning intensities. Optimize the controller to learn compression policies through direct interaction with the backbone rather than offline approximation.

4. Implement a physically correct dynamic pruning engine that removes transformer layers from the model's module hierarchy and maintains correct layer indexing for KV-cache alignment, eliminating implementation artifacts that cause incorrect pruning to degrade inference speed below the unpruned baseline.

5. Evaluate the end-to-end adaptive pruning system on held-out test prompts, reporting average inference speedup, perplexity degradation, per-source performance breakdown, zero-shot accuracy retention on unseen tasks, and controller overhead as a fraction of total latency, demonstrating that the framework achieves substantial speedup with bounded quality loss.

6. Conduct structured ablation studies that systematically isolate the contribution of each architectural component (the learned complexity router, hardware telemetry integration, and the reinforcement learning controller) by comparing the full system against variants that remove or replace each component, using both random and deterministic baselines to quantify the marginal benefit of adaptive decision-making.

1.5 Methodology in Brief

Figure 1.1 presents the methodology roadmap. The study employs a runtime-adaptive

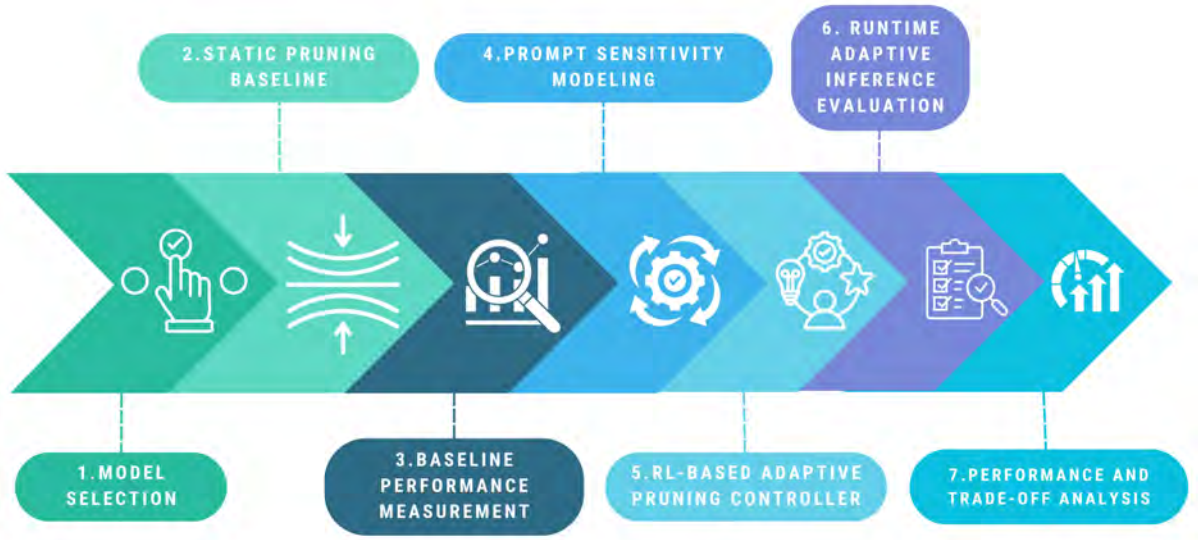


Figure 1.1: Methodology Roadmap

pruning framework controlled by a Reinforcement Learning (RL) agent to improve the efficiency of Large Language Models (LLMs) under dynamic resource constraints. Unlike traditional static compression methods, this approach enables per-prompt adaptive pruning decisions based on both system state and input characteristics. The research methodology is structured and systematically organized, progressing from baseline establishment to dynamic control and evaluation.

1. **Model Selection:** A medium-scale open-source LLM such as Llama-2-7B is selected as the base architecture. The model is chosen based on its open availability, strong performance on downstream tasks, and suitability for deployment in resource-constrained environments where full-precision models are impractical.

2. **Static Pruning Baseline:** A standard static pruning pipeline is implemented to establish a baseline. In this stage, a fixed level of pruning is applied to the model using state-of-the-art techniques such as SparseGPT or magnitude-based weight pruning, producing a compressed model with predefined sparsity. This baseline demonstrates what fixed offline compression can achieve and provides a reference point for the adaptive approach.

3. **Baseline Performance Measurement:** The statically pruned model is evaluated on standard benchmarks such as GSM8K, MMLU, and WikiText-2. Key metrics include inference latency, perplexity (PPL), and memory usage. This stage establishes a reference point for comparison with the proposed dynamic framework, quantifying the trade-offs inherent to static compression.

4. **Prompt Sensitivity Modeling:** A Learned Complexity Router (LCR) is developed to estimate the sensitivity of each input prompt to pruning. The router is trained using oracle labeling, where the performance differences between the dense unpruned model and pruned variants are directly measured. Rather than guessing how much a prompt can tolerate compression based on surface features, the LCR learns from actual loss gaps observed during inference. The LCR outputs a prompt sensitivity score ranging from 0 to 1, which helps determine how aggressively pruning can be applied without incurring significant quality loss.

5. **RL-Based Adaptive Pruning Controller:** A Double Deep Q-Network (DDQN) controller is implemented as the core decision-making module of the framework. The controller constructs a state vector using hardware telemetry (CPU utilization, available RAM, battery percentage, GPU availability, free VRAM, and GPU utilization), the prompt sensitivity score from the LCR, and early model signals such as hidden state norms and attention statistics from the first transformer layer. Based on this multi-modal state, the controller selects an optimal pruning action (for example, layer skipping at a particular intensity or attention head pruning at a particular sparsity level) for each prompt at inference time. The controller is trained using a reward function that balances inference speed improvement against quality preservation, measured via perplexity degradation.

6. **Runtime Adaptive Inference Evaluation:** The framework is evaluated in simulated resource-constrained environments by dynamically adjusting system conditions such as memory availability and compute load. For each prompt during evaluation, a pruning decision is made at runtime based on the current state, the pruned model is executed, and performance is measured and compared with the dense baseline. This process reveals how the controller adapts to changing conditions and prompt diversity.

7. **Performance and Trade-off Analysis:** Finally, the system's performance is analyzed by comparing the dynamic framework against static baselines. The evaluation focuses on latency reduction, perplexity degradation, speed–quality trade-offs, and RL reward progression across episodes. This analysis demonstrates the effectiveness of adaptive, per-prompt pruning in improving LLM efficiency under varying resource conditions, validating the core hypothesis that learned prompt sensitivity and hardware awareness outperform fixed compression strategies.

1.6 Scopes and Challenges

This research aims to develop a resource-aware hybrid compression model framework for LLMs, with the capability to dynamically adapt to compression strategies during inference based on the available system resources. By integrating different pruning techniques, followed by active monitoring by the controller agent, the system seeks to improve LLM efficiency without compromising performance. The framework will make LLMs more adaptable and deployable on resource-constrained devices such as local servers, mobile phones, and IoT devices. The key challenges and constraints are:

1. **Dynamic Compression Policy Selection:** It is technically challenging to design a dynamic system that adjusts the pruning decision based on real-time conditions (CPU load, memory, prompt sensitivity) without introducing significant computational overhead.

2. **Accuracy Preservation:** There is a possibility that active, vigorous dynamic compression will lead to a significant decline in the accuracy and performance of the model. This trade-off requires an appropriate strategic design of the control policy to be managed.

3. **Scalability of the Framework:** The framework must be able to scale properly to use large models and handle various tasks while being resilient when implemented in real-life situations. The variability of datasets and the challenge of generalizability make both devices and applications more complex.

4. **Accuracy of Simulation of Diverse Resource Environments:** The accurate simulation of the conditions in a resource-sparse environment is rather difficult to achieve. Simulation should consider dynamic resource states, i.e., variable memory, CPU usage, or battery power, which may influence performance.

5. **Evaluation and Benchmarking the Framework:** Exertion in real-world conditions regarding latency, memory utilization, and accuracy ought to be evaluated with utmost precision to benchmark the framework. The system will have to pass through a number of standard NLP benchmarks that will enable us to compare the performance of the static baseline while simultaneously considering the variability of the resource.

1.7 Thesis Structure

Chapter 1 is dedicated to providing the Introduction, which gives an overview of the background of Large Language Models and the computational challenges they pose, the motivation for developing a resource-aware adaptive inference framework, the problem statement concerning static compression limitations, the research objectives, a brief overview of the methodology adopted, and finally the scope and challenges of the research. **Chapter 2** is dedicated to providing the Literature Review, which gives an overview of the key preliminaries including pruning, quantization, knowledge distillation, and reinforcement learning, an extensive review of existing research on LLM compression and resource-aware optimization techniques, and finally a summary of the key findings from existing literature.

Chapter 3 is dedicated to providing an overview of the Impacts and Constraints, which gives an overview of the final functional and non-functional requirements and specifications of the proposed framework, the societal and environmental impacts of the research, ethical considerations, relevant standards, the project management plan, the risk management plan, and finally the economic analysis.
Chapter 4 is dedicated to providing an overview of the Proposed Methodology, which gives an overview of the design process and research workflow, the preliminary design and model specifications including dataset curation, data preprocessing, and oracle sensitivity labeling, the architecture of the Learned Complexity Router, the Reinforcement Learning Controller and Pruning Engine, a cross-method sensitivity analysis, and finally an ablation study of the proposed framework.
Chapter 5 is dedicated to providing an overview of the Result Analysis, which gives an overview of the performance evaluation of the LCR router and RL controller, an analysis of the design solutions including the reward function design, action space, and controller overhead, final design adjustments based on ablation studies, a statistical analysis using confidence intervals and Pareto analysis, and finally a comparison of the proposed framework against existing methods such as SparseGPT, Wanda, and LLM Pruner.
Chapter 6 is dedicated to providing an overview of the Conclusion, which gives a summary of the key findings, the main contributions of the research, the limitations of the proposed approach, and recommendations for future work.

Chapter 2
Literature Review

2.1 Preliminaries

Pruning involves eliminating unwanted components and those that are of the least significance in the model, thus reducing the size, complexity, and overload of a model. That makes a small model more effective, takes fewer resources, and is faster to use.

Resource-Aware systems are systems that can dynamically vary their operation as a reaction to the real-time availability of resources, e.g., battery power, memory, or computing capacity. With respect to LLMs, a resource-aware system might dynamically determine the compression methods at inference time based on the present system characteristics. It is relevant when operating the models in the real practical environments that have limited availability of resources, such as cell phones or edge stations.

Reinforcement Learning (RL) is a kind of machine learning whereby an agent learns to make choices based on the experience it can have upon interaction with its environment. Depending on its action, the agent will be rewarded or punished. It allows the agent to make the optimum choice when looking to do something in the end. RL finds many applications in areas such as decision-making systems, gameplay, or robotics, due to its capability of making sequential decisions in dynamic environments.

Knowledge Distillation is the ability to appropriately train a small model (the student) such that it behaves like a massive model (the teacher). The miniaturization will then mimic the actions of the bigger one with an intensive reduction of the resources needed. The result is a leaner, more effective model that does not forfeit most of the performance of the teacher model.

Quantization refers to the technique where weights of a model are coarsened (i.e., 32-bit floating point to 8-bit integer). This reduces the memory usage to a minimal degree, resulting in speeding up inference. Although this model may lead to reduced precision.

2.2 Review of Existing Research

In a comprehensive survey, Zhu et al. [1]investigated the critical field of model compression techniques in LLMs. The article's main goal was to provide a comprehensive overview of existing methods, challenges, and potential directions to improve efficiency and access to LLMs, especially where computational resources are limited. The authors

avoided embarking on new experimental research; instead, they aggregated evidence from a broad collection of existing research. These experiments used diversified data sets to assess compressed models' effectiveness. The performance of language models was measured using data sets like WikiText-2, C4, and PTB. For zero-shot task evaluations and reasoning skills tests, benchmarks like LAMBADA, PIQA, OpenBookQA, and GSM8K were used. The manuscript further pointed out the usage of large benchmark sets, like BIG-Bench, and test tools like the EleutherAI LM Harness, to ensure homogenous and comparable results. The survey covered a broad spectrum of LLMs, including models from the LLaMA series, OPT, BLOOM, and GPT series. The choice to review these models stemmed from their widespread use and the significant computational challenges they present due to their massive size. The paper systematically categorized the compression technologies into four main types: quantization, pruning, knowledge distillation, and low-rank factorization. It detailed various sub-methods, such as Weight-Only Quantization, Structured Pruning, and Black-box Knowledge Distillation. The performance of the compression techniques was measured using several metrics. The most prominent metrics were the compression and speedup ratios, representing the models' reduced size and improved inference time, respectively. Another notable metric used was the perplexity difference, which measures the loss in performance during language modeling after compression. The results demonstrated achievements such as weight-only quantization methods through GPTQ, which obtained a 3.24x speedup with a slight increase in perplexity for the OPT-175B model. Similarly, unstructured pruning methods like SparseGPT achieved a 50% compression ratio with minimal impact on performance. Thus, the survey effectively presented the considerable progress within the field, demonstrating that notable gains in efficiency and compression are indeed possible. In short, this survey forms a substantial and valuable manuscript. It adequately organizes and describes the existing state of LLM compression, making for a strong foundation for researchers and practitioners working toward more effective language models.

Based on another survey by Bai et al. [2], a wide array of efficiency methods was analyzed across the entire lifecycle of an LLM. These techniques were categorized into architecture design, pre-training, fine-tuning, inference, and system design, each with specific reported outcomes. In architecture design, methods focused on optimizing the core Transformer attention mechanism. Approximate attention techniques aimed to replace the standard mechanism, which has a quadratic time complexity of $O(T^2d)$ for time and $O(T^2)$ for memory, with more efficient alternatives. For instance, the Reformer model used locality-sensitive hashing to achieve a near-linear time complexity of $O(Td \log T)$. AFT-simple went further, eliminating dot-product operations for a linear $O(Td)$ complexity in both time and memory. Another approach, Modular Networks (MoE), enabled massive model scaling while maintaining constant computation. The GLaM model, with 1.2 trillion parameters, required half the computational power (FLOPs) for inference compared to GPT-3 by activating only 8% of its parameters for each input token. During the fine-tuning and inference stages, model compression techniques proved highly effective. Pruning, or creating sparsity, involves removing non-essential model weights. For example, the unstructured method, SparseGPT, could prune up to 60% of an LLM's parameters. Quantization reduced the numerical precision of model weights, shrinking storage and accelerating computation. The GPTQ method successfully compressed models to 3 or 4 bits per weight, while QLoRA enabled fine-tuning of these quantized models by integrating low-rank adapters. Dynamic acceleration methods optimize inference speed

based on the specific input. Speculative decoding used a smaller draft model to generate multiple tokens in parallel, which were then verified by the main model, resulting in a 2-2.5x speed-up in decoding without altering the output quality. At the system level, memory-efficient optimizers were developed to facilitate full-parameter fine-tuning on resource-constrained hardware. The MeZO optimizer, for instance, could fine-tune a 30-billion parameter model using only 55GB of GPU memory—a footprint equivalent to that of inference. A taxonomy of techniques for achieving resource-efficient LLMs diagram (Fig 2.1) is referenced below for a better understanding of the gist of how the efficient techniques are categorized based on LLMs.
Figure 2.1 presents a taxonomy of techniques for achieving resource-efficient LLMs. (Bai et al.).

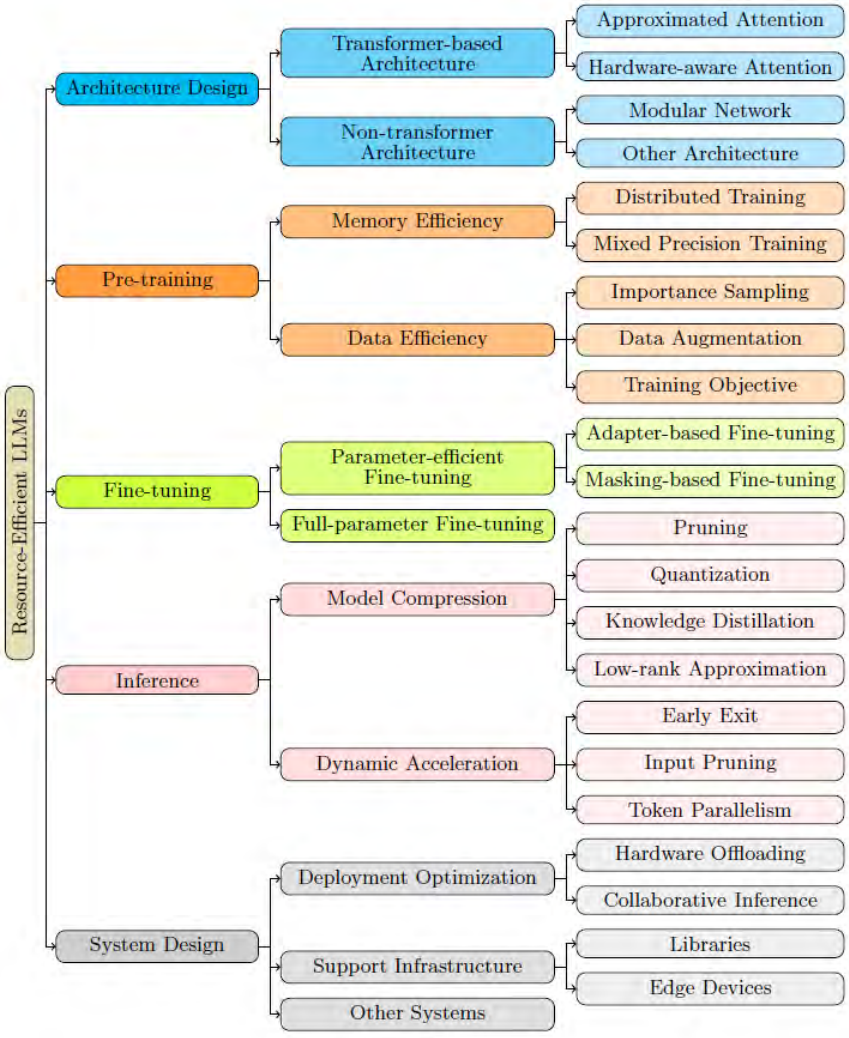


Figure 2.1: A taxonomy of techniques for achieving resource-efficient LLMs. (Bai et al.)

Frantar and Alistarh [9] suggested a novel one-shot post-training sparse self-supervised language model SparseGPT that was attempting to sparsify large-scale generative transformer language models, including OPT-175B and BLOOM-176B-SparGPT. The question was whether such a thing is achievable, i.e., whether it is possible to sparsify LLMs by 50-60-percent and then again not change its behavior with subsequent training and still be competitive in terms of perplexity, zero-shot task performance, which the paper

demonstrated with ease. Generative The developers of SparseGPT primarily compared SparseGPT to raw WikiText-2 and PTB, together with corresponding C4 subsets, and zero-shot evaluated LAMBADA, ARC (easy/challenge), PIQA and StoryCloze. Experiments in all tests were conducted on seed model family of OPT (125M175B) and BLOOM-176B all pruned with no fine-tune. SparseGPT is another solution to the challenge of layer-wise pruning since it is posing the issue as sparse regression and can be solved using a new approximate solution, which is scalable up to 100B+ parameter model. The solution suggests a speedy algorithm of estimating adaptive masks of pruning and layer-wise reconstructions based on approximations of Hessian. SparseGPT can do semi-structured pruning (2:4, 4:8), and integrating it with the other quantization schemes like GPTQ is straightforward, so one can end up with both pruning and 4-bit quantization at the same stage. Perplexity, zero-shot acc and inference speedup were applied as measures of performance. SparseGPT achieved proposed 50 percent weight sparsity on OPT-175B with minimal or no reduction in perplexity (8.35 to 8.21) in contrast to magnitude pruning and AdaPrune. Although SparseGPT maintained the high-level accuracy at 70.5 percentage average at five benchmarks under the zero-shot task, in contrast to the magnitude pruning at 31.1 percentages. The joint pruning + quantization (50%+4-bit) showed good performance on the perplexity as compared to the 3-bit quantized models implying that it performs well on the extremes of the compression. SparseGPT similarly does state-of-the-art on post-training pruning without any adaptivity or prompt-aware reasoning. The method also does not prune during training; it focuses on sparse weight pruning rather than activation pruning or latency-aware adaptation. However, SparseGPT serves as a decent reference of adequately scaled LLM compression, and it is the significant point of reference of the earlier effort at the study on static pruning of ours

Ma et al. [4] present LLM-Pruner, a framework for task-agnostic compression of large language models. One of their main goals was to perform this compression without heavily relying on the original training dataset, which was frequently proprietary or impractical to scale. They take a three-step approach to structural pruning. First, the framework analyzes the model automatically during the "Discovery" stage. "Coupled structures" are groups of neurons and weights that depend on one another. This will guarantee that all other components that rely on a deleted piece must also be deleted in order to maintain the model's integrity. The framework then evaluates the importance of each of these groups in the "Estimation" stage by looking at their gradient information, which shows how they affect the model's performance. Finally, once the groups of lesser importance are pruned, a fast "Recovery" phase uses Low-Rank Adaptation (LoRA) to fine-tune the compressed model and recover lost performance efficiently. The authors showed that LLM-Pruner on various models, LLaMA, and Vicuna, and able to strip 20% of the parameters, retaining close to a 95% of the performance of the original model after the careful tuning process with 50,000 data samples and then just in a relatively short timeframe of three hours. In spite of the novelty of its low resource pruning methodology, LLM-Pruner is a fixed optimization scheme. The amount of compression is not determined automatically, rather a developer have to do (specify manually) a fixed pruning ratio before starting the process. The framework carries out the specified instruction and produces a single, permanently pruned model designed for the chosen ratio. This design lacks the ability to adjust compression levels dynamically during operation. For instance, it cannot assess the memory capacity of a target device and then determine the most suitable pruning ratio automatically. As a result, when developers need to deliver models to devices with

varying hardware capabilities, they must repeat the pruning process manually for each ratio, which can be labor-intensive and inefficient. Moreover, the authors point out that higher pruning levels, such as 50%, can cause remarkable drops in performance, underscoring the limitations of a fixed-ratio method. Because the compression level is set in advance and not adapted in real time, the model produced by the framework is locked to the specified ratio. This static nature means that the system cannot fine-tune compression based on live resource constraints, making it impractical for flexible, multi-device deployments. High pruning rates can lead to a major fall in model performance, which highlights the difficulty and rigidity of a fixed-ratio approach.

Liu et al. [12] introduced PAT (Pruning-Aware Tuning), a structured pruning framework to integrate channel sparsification with the fine-tuning of LLMs. The authors reduced model size and accelerated inference at the same time, holding or even improving accuracy—averting degradation typically associated with post-hoc pruning.The team tested the models with the following optimizations for testing: LLaMA2 (7B, 13B), Gemma (2B, 7B), and Yi-1.5-34B optimized on the LaMini-Instruct dataset (1M samples after downsampling). Zero-shot generalization was tested on 14 downstream datasets like MMLU, PIQA, WinoGrande, OpenBookQA, and BOOLQ for common sense reasoning, QA, and logic tasks. Models were tested with varying pruning ratios (20–30%). Fine-tuning involved Alpaca templates, cosine learning rate scheduling, and PEFT setups (e.g., LoRA/DoRA). Methodologically, PAT utilized Hybrid Sparsification Modules (HSMs) inserted between FFN and attention blocks. The modules applied a learnable sparsity mask (USM) and Hybrid-Identity Operator (HIO) to maintain the gradient flow across learning that guides which channels to prune. An Identity Loss was formulated to isolate scaling and rotation during pruning to be more resilient. The pruning message was conveyed by RMSNorm in a way that structural reduction downstream was done without sacrificing accuracy. Perhaps most significantly, PAT pruning and tuning were accomplished simultaneously end-to-end. On the metric side, PAT was assessed in terms of accuracy, memory, latency, and the number of trainable parameters. PAT achieved 60.02% accuracy on LLaMA2-7B at 25% pruning, higher than LoRA-64 (58.76%) and significantly higher than SliceGPT and LLM-Pruner with similar pruning. The 30% pruned PAT had 98% unpruned model accuracy. It also had a 1.33× latency speedup, fit into GPU memory with larger batch sizes, and needed fewer parameters than fine-tuning the full LoRA. PAT shows how sparsity and performance are not in conflict when pruning knows about fine-tuning. Its contribution is primarily to represent pruning as a tunable, differentiable part of tuning. It does not have quantization or runtime adaptability, but PAT establishes a solid structured compression foundation, so it is extremely well-suited for hybrid approaches that prefer trade-offs between efficiency and accuracy.

Chen et al. [11] presented DLP, a dynamic layerwise pruning algorithm for accelerating LLM inference, aimed at reducing compute latency and memory on the fly with no sacrifice in output quality. Its core goal was to enable at runtime choosing transformer layers to skip based on input complexity and target latency, avoiding the inflexibility of static pruning and the expense in generalization over tasks or prompts. To evaluate DLP, the authors utilized both reasoning and generative tests. Language model testing was carried out using WikiText-2, and downstream testing was done with MMLU and OpenBookQA. Tested models included LLaMA-7B, OPT-6.7B, and Mistral-7B, which offered ample testing on dense and decoder-style models. Inputs were preprocessed into fixed prompt templates that included mixed instruction-following and reasoning tasks. Train-

ing included standard autoregressive loss, sparsity scheduling, and pruning fine-tuning. The method introduced an importance score layer in terms of the activation norm as a signal to tell which transformer blocks to prune for a given input. DLP then used a scaling scheduler to scale between target latencies and pruning ratios dynamically during inference. Importantly, no retraining from scratch or access to external gating predictors is required. Fine-tuning was used to achieve stabilization of the pruned model for consistent performance. For performance measurements, DLP was quantified as inference latency, perplexity, F1/Accuracy, and token throughput. For LLaMA-7B, DLP achieved 2.1× speedup and 35% lower latency at a ¡1% loss of accuracy on QA tasks. Perplexity loss was minimal on WikiText-2 (from 5.4 to 5.6), much less than static pruning baselines like LayerDrop and uniform layer skipping. DLP was also shown to generalize strongly to zero-shot tasks without additional tuning. Even though successful, DLP did not include quantization or memory profiling in decision-making logic. Pruning was limited to whole layers, and more detailed optimizations like attention head or MLP block pruning were not investigated. More extension of DLP to multi-objective compression, hardware-specific scheduling, and hybrid architecture was suggested by the authors. This study directly validates our research direction, illustrating that runtime-aware pruning enhances LLM inference speed. DLP's simplicity in architecture, strong performance, and capacity to generalize across tasks validate dynamic, structured compression within hybrid optimization frameworks.

Liu et al. [5] proposed a promising method of optimization of the LLM. The idea behind the subject of study in the research article, named "RAP: Runtime-Adaptive Pruning for LLM Inference" was to grapple with the massive computation and memory demands that usually come with LLM. The main target was to create a self-optimizing system in real-time with the dependency of the available memory space which, at the same time, could adapt itself to the individual requirements of the user queries. The development of the methodology and the testing performed by the authors included several datasets. More fine-tuning was then done on the C4 and WikiText large datasets. A full suite of benchmarks was used to test the downstream target-task performance, including the ones of RACE, BoolQ, PIQA, SIQA, WinoGrande, HellaSwag, ARC and OBQA. This broad set of datasets adequately demonstrated the robustness and effectiveness of their procedure across language-understanding and reasoning tasks. Their proposed RAP framework was implemented using reinforcement learning (RL). This was evaluated on large language models such as Llama 2 7B, Llama 2 13B, and Mistral 7B. The use of an RL-based strategy was necessary to enable adaptation and determine the most useful model components throughout the pruning process. The RL agent was used to test the space-efficiency-tradeoff since it took into account the learning dynamics of feed-forward networks (FFNs), with high parameter density and attention layers performing the most computationally expensive KV-cache operations. Their study has demonstrated a promising insight on the premise that their empirical results were substantially encouraging. Multiple downstream effects based on accuracy were analyzed, and a language model evaluation standard perplexity was adopted to analyze the effectiveness of RAP method. Furthermore, the researchers calculated the compression ratio to find out the possible compression level. Their RAP-Llama-2-7B implementation achieved an impressive compression ratio of 2.25x with a much-marginal low average reduction in accuracy of only 2.3% on the different tasks evaluated. This result demonstrates that their intention to build an efficient and universal pruning method succeeded and exceeded the levels

of performance of prior methods of static pruning. Specifically, present study contains a great and practical contribution to artificial intelligence. RAP is presented as an efficient way to deploy more complex LLM in low-resource environments, and therefore to make such advanced technologies more accessible. The adaptive nature that exists in RAP is an important improvement over previous methods as it represents a more versatile and intelligent way of reducing the cost of computation from massive AI models

Federici et al. [7] presented a predictor-free method called DIP (Dynamic Input Pruning), and its cache-conscious variant (DIP-CA), to speed up inference of large language models (LLMs) on low-memory mobile phones. The primary objective was to mitigate the performance bottleneck due to limited memory bandwidth, especially since newer LLMs such as Mistral, LLaMA 3, and Phi-3 use non-ReLU activations (e.g., SwiGLU), not sparsely activated in nature. The authors evaluated DIP on WikiText-2 with language modeling and downstream reasoning with MMLU (5-shot) using models such as Phi-3-Medium, Phi-3-Mini, LLaMA3-8B, and Mistral-7B, all of which were quantized to INT4. Preprocessing involved tokenized calibration datasets such as SlimPajama, optionally followed by light fine-tuning with LoRA adapters. Interestingly, the authors introduced a hardware simulator to emulate memory and flash read dynamics for estimating latency, memory, and throughput. DIP prunes low-magnitude input activations according to a top-K threshold, evading complex predictors (compared to DejaVu). Its cache-aware variant DIP-CA is also biased towards pruning mask weights that are in the DRAM cache, further improving cache hit rate and reducing latency. Both methods were quantization and LoRA tuning compatible. Against baselines like SparseGPT, DejaVu, CATS, and even GLU oracle pruning, DIP achieved state-of-the-art trade-offs: on Phi-3-Medium, DIP-CA achieved 46% memory reduction and 40–93% throughput gain at ¡0.1 perplexity loss. For sparsity 50%, DIP+LoRA achieved MMLU accuracy of 77.39% (vs. 78.14% dense) and WikiText perplexity of 4.62 (vs. 4.29 dense), 2–7% better than others. This work uncovers basic limitations of traditional pruning methods on SwiGLU models and avoids costly retraining. It demonstrates that hardware-aware sparsity combined with cache optimization can support local LLM inference without performance loss. The framework is, however, limited to inference only and does not extend to pruning at training time or other hybrid compression modes like distillation

Elias Frantar et al. [8] introduced GPTQ, a one-shot, second-order post-training quantization method for compressing extremely large-scale language models like OPT-175B and BLOOM-176B to 3–4 bits per weight with minimal loss of accuracy. The most critical goal was to make it possible to carry out inference with multi-billion-parameter models on a single GPU, with no retraining or access to labeled data. Authors evaluated GPTQ on OPT and BLOOM model families across conventional generative benchmarks such as WikiText2, PTB, C4, and zero-shot tasks such as LAMBADA, PIQA, StoryCloze, and ARC. They used a tiny calibration set (128 randomly sampled 2048-token sequences from C4) and quantized whole Transformer models, including attention and MLP blocks. GPTQ extends prior methods like RTN and ZeroQuant by introducing approximate second-order information in the form of a low-rank inverse Hessian to guide weight updates during quantization. The method quantizes layer-wise with Cholesky decomposition to avoid expensive matrix inversions, enabling both precision and scalability. By combining this with lazy batched updates and group-wise quantization, GPTQ is stable and minimizes numerical drift even for 175B-parameter models. In terms of outcomes, GPTQ quantized OPT-175B to 4-bit precision in 4 hours on a single A100

GPU with 8.37 perplexity compared to 8.34 FP16 on WikiText2. In contrast, RTN fell to 10.54. Even at 3 bits, GPTQ preserved performance (8.68 perplexity), while RTN completely collapsed. On zero-shot tasks like LAMBADA and PIQA, GPTQ at 4 bits exceeded FP16 accuracy, and even 3-bit models preserved good generalization. Latency was improved by up to 4.5× on A6000 GPUs via reduced memory I/O, even in the absence of activation quantization. While GPTQ is weight-only, not activation, and static, not dynamic, GPTQ sets a new state-of-the-art for low-bit, high-fidelity quantization at massive scale. For our research, GPTQ serves as a valuable quantization module and baseline, showing that it is possible to use aggressive compression, and now it can be incorporated into dynamic, adaptive, and hybrid systems.

Lin et al. [6] introduced AWQ, an emerging low-bit weight-only quantization scheme, that sought to compress large language models (LLMs) for on-device inference. The initial objective was to reduce memory footprint and inference latency without employing quantization-aware training (QAT), which is not feasible for billion-parameter models. AWQ aimed to preserve the generalization capacity of LLMs through discovering and scaling a few important weight channels based on activation statistics and not using backpropagation or weight size. The researchers used WikiText-2, MBPP, and GSM8K for math evaluation, coding, and language modeling. OPT (1.3B–30B), LLaMA (7B–65B), Llama-2 (7B–70B), Mixtral, and Mistral models were utilized. Multimodal, instruction-tuned Vicuna, VILA, and OpenFlamingo-9B models were also tested. Preprocessing involved the collection of average per-channel activation scales from a small calibration set and the use of grouped quantization (INT3/INT4 group size 128). AWQ's strategy involved scaling key channels before quantization to minimize their relative error without the need for mixed-precision formats. It involved no reconstruction, no retraining, and domain generalization was maintained. For real-time deployment, the authors presented TinyChat, an inference engine optimized with fused kernels and SIMD-aware weight packing, achieving real-world latency gains. Evaluation criteria were perplexity (PPL), accuracy, and token throughput. On WikiText-2, AWQ outperformed GPTQ and RTN across the board, reducing PPL from 23.54 to 11.92 on OPT-6.7B with only 1% salient weights scaled. On COCO and Vicuna benchmarks, AWQ attained near-lossless performance with INT4 quantization. TinyChat achieved up to 3.3× speedup over HuggingFace's FP16 baseline on GPUs and could run Llama-2-13B on 8GB memory laptops at 30 tokens/s.The work highlighted that AWQ does not support activations or layer skipping and is specifically meant only for weight quantization. Future work includes combining AWQ with pruning and additional support for activation quantization. Despite this, AWQ demonstrates strong, hardware-optimized quantization with minimal in the way of calibration data and strong generalization. Its modularity and on-device suitability strongly support the need for saliency-aware, resource-adaptive quantization in hybrid compression systems.

A new scheme of increasing the efficacy of computational resource utilization through LLMs was proposed by Jiang et al. [3]. They wanted to make computing and memory expenses comparatively cheap by means of utilizing LLMs, particularly when resources are limited. Based on the premise of the computational resources, they proposed a dynamic inference algorithm based on the observation that tasks or data differ in their needs of computational resources namely D-LLM. Most of the benchmarks used were very many to check effectiveness of their solution. The used data sets were divided into three major

sets, including question-answering and summarization, mathematical problem-solving, and common-sense reasoning, which have Alpaca and SAMSum, GSM8K, and MaWPS, BoolQ, PIQA, SIQA, OBQA, and MMLU, respectively. The variety of these tests allowed in-depth measurements of D-LLM's performance across a broad range of linguistic tasks. The logic of their study lay on the grounds of utilizing well-established LLMs, LLaMA2-7B, and LLaMA3-8 B. The rationale of applying the models was partly based on the fact that they were large and hence the high cost of resources that LLM large, which this research strives to solve, have. A unique feature of their design was the inclusion of a "dynamic decision module" for each of the transformer layers of the model, which decides how to process a given token. The property was merged with a "KV-cache eviction strategy," which improved memory management among the different layers, thus maintaining alignment with traditional acceleration techniques. In order to avoid the intensive retraining costs, the authors supplemented their approach with the Low-Rank Adaptation (LoRA) fine-tuning method. The findings signified astounding achievement of the objectives posited in the paper.The D-LLM model achieved significant reductions in computational costs, measured in terms of Floating Point Operations per second (FLOPs), with reductions by up to 45% for question-and-answer and math tasks, and up to 50% for common-sense reasoning tasks, while ensuring similar performance levels. Relative to the MaWPS and OBQA data, the D-LLM model achieved baseline performance metrics with only 30-40% of the computational FLOPs commonly used. Measuring success involved metrics including accuracy measures for math and reasoning tasks and perplexity (PPL) metrics for question-and-answer and summarization activities. Considering the significant reduction in computational demands and memory use, the manuscript presents a highly pragmatic and practical model expected to improve the efficiency and applicability of LLM.

Yang et al. [10] proposed a benchmarking suite, named LLMCBench, by which compression methods of LLMs are evaluated by systematically measuring the performances of LLMs on realistic deployment environments. The goal of this paper was to go beyond the fragmented state of LLM compression benchmarks by providing an integrated platform for performance, efficiency, generalization, trustworthiness, and hardware compatibility. The authors have chosen six commonly-used LLMs (LLaMA (1/2/3), Vicuna, OPT, ChatGLM) and compared SparseGPT, Wanda, LLM-Pruner, GPTQ, SmoothQuant, AWQ, and OmniQuant with each other. They have been benchmarked on 11 datasets (MMLU, PIQA, WinoGrande, ARC, HellaSwag and WikiText2) consisting of both generative and reasoning tasks. For Trustworthiness, TruthfulQA and AdvGLUE were added as well. LLMCBench broke evaluation down into six tracks, which are the following: compression performance generalization training consumption inference consumption hardware acceleration trust. Tightening: each track was based on hard metrics. For instance, as Track 1 did accuracy preservation on knowledge and inference tasks (OMperf), and Track 4 was playing around with memory, MACs, and model size. The benchmark also implemented real world deployment to libraries such as TensorRT-LLM, vLLM and MLC-LLM. Quantization strategies have surpassed sparsity strategies in most tracks, and in particular in trustworthiness (OMtrust ? 97 for quantizers vs. ! 90 for sparsity strategies). GPTQ and AWQ generalized the best (OMgen ¿93) and LLM-Pruner was the best sparsity technique for inference memory as well as structured acceleration. Quantization did surprisingly better as far as knowledge retention is concerned, however, sparsity maintained inference

efficiency. Activation quantization (i.e., SmoothQuant) failed to generalize. The paper highlighted that accuracy is not the only deployment consideration - training cost, hardware acceleration and trust are no less important. LLMCBench gives a first-of-its-kind comparative overview of LLM compression techniques, and establishes a good empirical foundation for future research. However, it does not focus on the use of dynamic compression techniques or KV-cache pruning which is essential to on-device inference. This limitation is in perfect accordance with our research motivation to bring runtime adaptivity into the compression pipeline. As a result, this paper not only presents a good reference benchmark but it also indirectly demonstrates the novelty of the adaptive prompt-aware hybrid models and maintains the inference efficiency. Other techniques on quantization (e.g. SmoothQuant) have poor generalizations. The paper highlighted that accuracy is not the only deployment consideration - training cost, hardware acceleration and trust are no less important. LLMCBench gives a first-of-its-kind comparative overview of LLM compression techniques, and establishes a good empirical foundation for future research. However, it does not focus on the use of dynamic compression techniques or KV-cache pruning which is essential to on-device inference. This limitation is in perfect accordance with our research motivation to bring runtime adaptivity into the compression pipeline. As a result, this paper is not only an excellent benchmarking reference but also indirectly demonstrates the novelty of adaptive timely anticipative hybrid models.

2.3 Summary of Key Findings

This research aims to solve one of the most significant issues that prevent the mass use of LLMs: the inability of modern compression algorithms. Albeit effectively reducing LLMs' huge CPU and memory demands, pruning, quantization, and knowledge distillation are primarily static, one-time solutions. This generic approach thus ends up with de facto models locked away in a permanently compressed state and never allowed to respond dynamically to changing environmental demands under which practical systems have to operate, e.g., memory availability, CPU availability, battery state, or complexity of user request. Such an intrinsic lack of flexibility greatly limits the implementation of large, sophisticated LLMs on devices with limited resources, e.g., cell phones, local servers, and the Internet of Things (IoT) devices. Due to a significant gap in doctrines, the present study suggests developing a new resource-efficient hybrid model that selectively and methodically decides on the most desirable balance between the compression methods during inference. Second, the most notable is a better Controller Agent developed with great consideration for real-time operation. The controller not only reads the telemetry of the devices, e.g., CPU consumption, memory size, battery level, etc., but also estimates the quality of the prompts. Based on this real-time data, the controller will go into an optimal compression strategy, which will be obtained by dynamically varying the pruning and quantization rates correlated with the particular inference request. This kind of modern technique is comparable to the action of modern video games when they detect and adjust the parameters automatically accordingly to achieve the best performance based on the hardware on which the system is being used by the user, which eases the favoring of the system whereby, the AI can adapt to the user instead of the reverse being true. The track in which the present paper is written is system-focused

towards developing and testing the target system. Phase one would choose a strong, de-restricted, open-source foundation model, e.g., Llama 3 or Phi-3. They will present a bottom-line performance metric, implemented using a fixed hybrid compression path using state-of-the-art methods, i.e., Wanda to prune the models and Activation-aware Weight Quantization (AWQ) to quantify them. Next, a dynamic system will be built, with a dynamic controller consisting of a rule-based but increasingly more advanced RL learner capable of learning complex optimizing approaches. It is necessary to test comprehensively strong prior understandings and instruction-execution rules to measure accuracy, delay of inferences, and memory usage with constraints of virtual hardware. It is noted that the overall problematic technical issues that the current study gallops with are the necessity to come up with a dynamic system of policy selection having the aim to minimize unnecessary computing expenses, the top priority about retaining the model accuracy even subject to thorough compression, as well as the anticipation of the system to be scalable as well as robust in regards to challenging the system with diverse tasks and hardware configurations. The literature review also justifies the direction the study is taking by citing that even though different successful static compression techniques have been invented, namely SparseGPT, GPTQ, and AWQ, they collectively lack runtime flexibility. The practicality and efficiency of the dynamic pruning approach were confirmed in recent developments of RAP and DLP, which justified the rationality and efficiency of the adaptability awareness approach at runtime and provided a solid basis for the current research effort. This research attempts to contribute to the field in that it attempts to build a hybrid solution that can manage various tools in compression in a unified intelligent control system, hence creating an efficient and feasible solution to scale large, high-performance language models on standard hardware.

2.4 Research Gap and Contribution

Despite significant advancements in model compression techniques such as pruning, quantization, and distillation, most existing approaches remain static and are applied prior to deployment. These methods do not adapt to variations in prompt complexity or real-time system constraints during inference.

While recent work has explored dynamic pruning strategies, these approaches typically rely on heuristic-based complexity estimation and do not incorporate learned sensitivity modeling or multi-modal system state inputs.

This research addresses this gap by introducing a unified adaptive inference framework that integrates oracle-trained sensitivity prediction, reinforcement learning-based decision-making, and real-time hardware awareness to enable dynamic, per-prompt optimization of large language models.

Chapter 3

Impacts and Constraints

3.1 Final Specifications and Requirements

SPRINT operates as a five-stage pipeline in which each stage produces artifacts consumed by the next. This dependency structure means that quality constraints propagate throughout the system: a poorly curated dataset leads to noisy oracle labels, noisy labels weaken the prompt-sensitivity router, and a weak router subsequently degrades the quality of the signal provided to the reinforcement learning controller. For this reason, both functional requirements, describing what the system must accomplish, and non-functional requirements, describing the quality constraints under which it must operate, were defined prior to implementation and later verified empirically.
Table 3.1 summarizes the principal functional requirements together with the evidence used to verify their satisfaction.

Table 3.1: Functional requirements of the proposed SPRINT framework.

ID	**Requirement**	**Verification**
FR-1	Assemble a balanced, multi-domain prompt dataset from at least five public benchmarks	10,000 prompts (5 × 2,000), audited CSV output
FR-2	Generate per-prompt oracle sensitivity labels using dense and sparse backbone inference	Oracle CSV with composite normalized labels in the range [0,1]
FR-3	Train a lightweight prompt-sensitivity router that predicts oracle labels with Spearman $\rho \geq 0.70$	Test $\rho = 0.797$, 95% CI [0.779, 0.817]
FR-4	Implement a DDQN controller that selects per-prompt pruning actions from a 17-action discrete space	Converged policy with positive average reward
FR-5	Achieve measurable inference speedup ($\geq$ 10%) through physical structural pruning	32.0% average speedup on 2,000 held-out test episodes
FR-6	Maintain bounded quality degradation under controller-selected pruning	Token-weighted perplexity ratio monitored per episode
FR-7	Keep controller overhead below 5% of total inference latency	17.77 ms overhead, equivalent to approximately 2.0% of total latency
FR-8	Ensure that all pruning operations are fully reversible between episodes	Model restoration verified after every episode

Among these requirements, FR-3 is particularly important because the router provides the primary signal that guides adaptive pruning. The target threshold was set at Spearman $\rho \geq 0.70$, while the final router achieved $\rho = 0.797$. This margin is not merely cosmetic; rather, it indicates that the router can consistently distinguish pruning-sensitive prompts

from pruning-tolerant prompts and can therefore support meaningful downstream control decisions.
Table 3.2 presents the principal non-functional requirements that governed the design and deployment constraints of the system.

Table 3.2: Non-functional requirements of the proposed SPRINT framework.

ID	Requirement	Constraint
NFR-1	Run entirely on consumer-grade hardware without cloud dependencies	Single RTX 5090-based workstation, as summarized in Table 3.3
NFR-2	Ensure reproducibility so that all experiments are deterministic under fixed seeds	Seed = 42 for all random operations
NFR-3	Use only publicly available datasets and open-weight models	Hugging Face-hosted benchmarks and Llama-family models
NFR-4	Keep LCR router inference latency below 30 ms on CPU	Measured at approximately 28 ms on an AMD Ryzen 9 9950X
NFR-5	Support the selected 7B backbone across all pipeline stages	Llama-2-7B supported end-to-end
NFR-6	Maintain full version control of source code and generated artifacts	Git repository with complete experimental history

The software stack used to implement the system consists of Python 3.9 or later, PyTorch 2.5 with CUDA 12.1, the Hugging Face *Transformers* and *Datasets* libraries, `psutil` for system telemetry, and NVML for GPU monitoring. A single hardware configuration was used, selected to match the memory requirements of the Llama-2-7B backbone.

Table 3.3: Hardware configuration used for all experiments.

Component	Experimental workstation
GPU	NVIDIA RTX 5090 (32 GB VRAM)
CPU	AMD Ryzen 9 9950X (16 cores, 32 threads)
RAM	64 GB DDR5
Storage	1 TB SSD + 1 TB HDD
OS	Windows 10/11

All experiments, including dataset curation, oracle labeling, LCR training, RL controller training and testing, and ablation studies, were executed on this single workstation using Llama-2-7B. The model's memory footprint in float16 is approximately 14 GB, and the RTX 5090's 32 GB VRAM provides ample headroom together with key-value cache allocation during generation. This remains a consumer-grade desktop configuration and therefore stays consistent with the project's accessibility goals.

3.2 Societal Impact

SPRINT addresses a practical and increasingly important barrier to the use of large language models: the cost of running billion-parameter models locally on consumer hard-

ware. In many real-world settings, users who require privacy cannot always rely on cloud-based APIs. A lawyer reviewing confidential contracts, a physician summarizing patient notes, or a developer handling proprietary source code should not be forced to choose between the benefits of language-model assistance and the confidentiality of their data. By making local inference faster and lighter through adaptive pruning, SPRINT offers a third option in which computation remains on the device while still delivering useful model outputs.

Privacy constitutes the most immediate societal benefit of the proposed framework. The pruning controller adapts to available hardware conditions in real time, whether the system is deployed on a desktop workstation with a dedicated GPU or on a battery-powered laptop. In this setting, the user does not need to manually configure pruning profiles. Instead, the framework reads variables such as CPU load, available VRAM, and battery state, and then selects a pruning intensity appropriate to the current environment.

At the same time, broader accessibility must be considered alongside broader risk. Any system that makes language-model inference cheaper and more widely available can also make large-scale text generation easier. This raises concerns related to misinformation, plagiarism, and social-engineering misuse. However, SPRINT does not alter the underlying generative capability of the backbone model; it changes only how efficiently the model runs and where that model can be executed. Moreover, higher pruning intensities reduce output fidelity relative to the dense baseline, which means that the framework does not amplify model quality so much as it improves deployment flexibility.

The accessibility implications are also substantial. Because the framework runs on a consumer GPU workstation built around an RTX 5090, students, researchers, and developers working outside enterprise environments can still experiment with runtime-adaptive inference locally. In this sense, the proposed system contributes to a more inclusive AI ecosystem by lowering the entry barrier for local experimentation while preserving user control over sensitive data. The codebase, dataset pipeline, and training artifacts are also described as open-source in the project brief, further strengthening the accessibility of the work.

3.3 Environmental Impact

The environmental implications of large language model research are closely tied to energy consumption. From the outset, this project prioritized a design that could run on a single consumer-grade GPU rather than relying on a multi-GPU cluster. In practice, the full system was developed and evaluated on an RTX 5090 with a 575 W thermal design power. The oracle-labeling stage was the most computationally demanding component of the pipeline. It required approximately 30,000 forward passes through Llama-2-7B, corresponding to 10,000 prompts evaluated under one dense and two sparse configurations. Nevertheless, this stage is a one-time cost. Once the labels are generated and stored, they can be reused during router development and subsequent experiments without repeating the same expensive computation.

The LCR router is comparatively lightweight. Built on an 11.3 million-parameter BERT-mini backbone, it trains quickly on the RTX 5090 workstation. Relative to the enormous computational budgets associated with pre-training modern LLMs, this cost is negligible. Reinforcement learning for the controller remains more expensive than router training, but it was still completed on a single GPU. Across development, ablation studies, and

testing on the experimental workstation, the total GPU usage for the entire project is estimated at 60–90 GPU-hours, which remains small when compared with the hundreds of thousands of GPU-hours associated with pre-training large open-weight models.
At inference time, the framework contributes positively to energy efficiency. Physical layer removal reduces the number of matrix multiplications required during generation, and when the controller selects high-sparsity actions, the transformer executes substantially less computation than the dense baseline. The measured 32.0% average speedup therefore translates into reduced GPU active time per prompt. On desktop systems this implies lower cumulative energy consumption, while on battery-powered devices it can contribute directly to longer operating time between charges. Consequently, the framework aligns well with the broader goal of environmentally responsible AI deployment by reducing both hardware requirements and runtime energy demand.

3.4 Ethical Issues

The ethical considerations of this project can be grouped into three areas: data provenance, model licensing, and the consequences of making local LLM inference more efficient. These issues were considered throughout both the data-construction and system-design phases.
All five benchmark datasets used in the study are publicly available and were originally released for research purposes. GSM8K, MBPP, MMLU, BoolQ, and WikiText-2 are well-established benchmarks that are commonly used in academic work. The dataset construction pipeline streamed these sources through the Hugging Face *Datasets* API and applied only minimal source-specific formatting required for structural consistency. No personally identifiable information was introduced, and no manual annotation involving human participants was required.
Model licensing was also reviewed carefully. The backbone model, Llama-2-7B, was used from its publicly released checkpoint under its community license for academic research. The prompt-sensitivity router backbone, `prajjwal1/bert-mini`, is available under the Apache 2.0 license. All models were used in accordance with their published licensing terms, and the project did not involve scraping, reverse engineering, or any attempt to bypass access restrictions.
An additional ethical concern arises from the possibility that faster local inference may increase the volume of AI-generated text, including harmful or misleading content. This concern is valid, but it is not unique to this framework. SPRINT does not endow the model with new knowledge or new generative abilities; rather, it adjusts computational allocation according to prompt sensitivity and system conditions. The controller operates on workload characteristics rather than semantic intent, meaning that it does not inspect or optimize for the social content of the generated output. As a result, the ethical risk profile remains comparable to that of any other efficiency-oriented deployment improvement.
No human subjects were involved in this research, and all evaluation procedures were automated, deterministic, and benchmark-based. This reduces privacy risk and avoids the ethical complications associated with collecting or labeling human participant data.

3.5 Standards

Although no single formal regulatory standard directly governs exploratory research on neural-network pruning for LLM inference, the project followed widely accepted community standards related to reproducibility, comparability, and methodological rigor. These standards shaped both the evaluation strategy and the software-engineering practices used during implementation.

For quantitative evaluation, perplexity was computed using token-weighted cross-entropy loss, following standard language-modeling practice. Zero-shot accuracy on benchmarks such as BoolQ and MMLU was measured using their standard benchmark formulations. Correlation and regression analyses were reported using conventional statistical metrics including Spearman rank correlation and the coefficient of determination, and confidence intervals were estimated by bootstrap resampling. These choices make the reported findings easier to compare with prior literature and improve the credibility of the analysis.

The software implementation also followed common engineering conventions. The codebase adheres to Python packaging norms, including PEP 8 style, the use of type hints, and docstrings for public functions. The pruning engine interfaces with PyTorch modules through public APIs rather than relying on undocumented internals, which improves maintainability when upstream libraries evolve.

Reproducibility was enforced systematically. All stochastic operations used a fixed random seed of 42. Dataset splits were stored explicitly, and metadata files preserved configuration details such as the backbone identifier, sparse settings, sequence length, normalization bounds, and runtime statistics. Accordingly, given the same environment, checkpoint, and seed, the core results reported in this thesis can be regenerated from the recorded artifacts.

3.6 Project Management Plan

The project followed a five-phase management plan spanning approximately sixteen weeks. Because SPRINT is organized as a sequential pipeline, later stages depend directly on the outputs of earlier stages. This meant that phases could not overlap arbitrarily, although parallel work was still possible within each phase wherever dependencies allowed. Table 3.4 presents the overall project timeline, major activities, and expected deliverables.

Table 3.4: Project timeline for the development of SPRINT.

Phase	Duration	Activities	Deliverables
1	Weeks 1–3	Literature review, architecture design, hardware setup, and environment configuration	Design document and software environment
2	Weeks 4–6	Dataset construction pipeline, quality-audit scripts, and benchmark source integration	`Oracle_dataset.csv` containing 10,000 prompts and associated audit reports
3	Weeks 6–9	Oracle labeling pipeline, composite sensitivity scoring, normalization, and LCR architecture training	`oracle_lcr_labels.csv` and a trained MiniBERT router
4	Weeks 9–13	DDQN controller development, pruning-engine implementation, reward-function design, and full-pipeline integration	Trained RL policy and integrated end-to-end system
5	Weeks 13–16	Ablation studies, zero-shot evaluation, report generation, and thesis writing	Ablation reports, test reports, and the completed thesis document

The phased development approach ensured systematic progress, where each stage built upon the outputs of the previous phase. This structured workflow minimized integration risks and allowed parallel development of independent components such as the pruning engine and reinforcement learning controller.

The timeline also enabled iterative refinement through feedback cycles, ensuring that critical issues such as cache misalignment and reward instability were resolved during development.

Figure 3.1 complements Table 3.4 by visualizing how the five phases were distributed across the sixteen-week schedule. The chart makes the sequential dependencies between phases easier to interpret, while also showing where limited parallel work was feasible within the broader plan. Together, Table 3.4 and Figure 3.1 provide both a tabular and visual view of the project-management structure used for SPRINT.

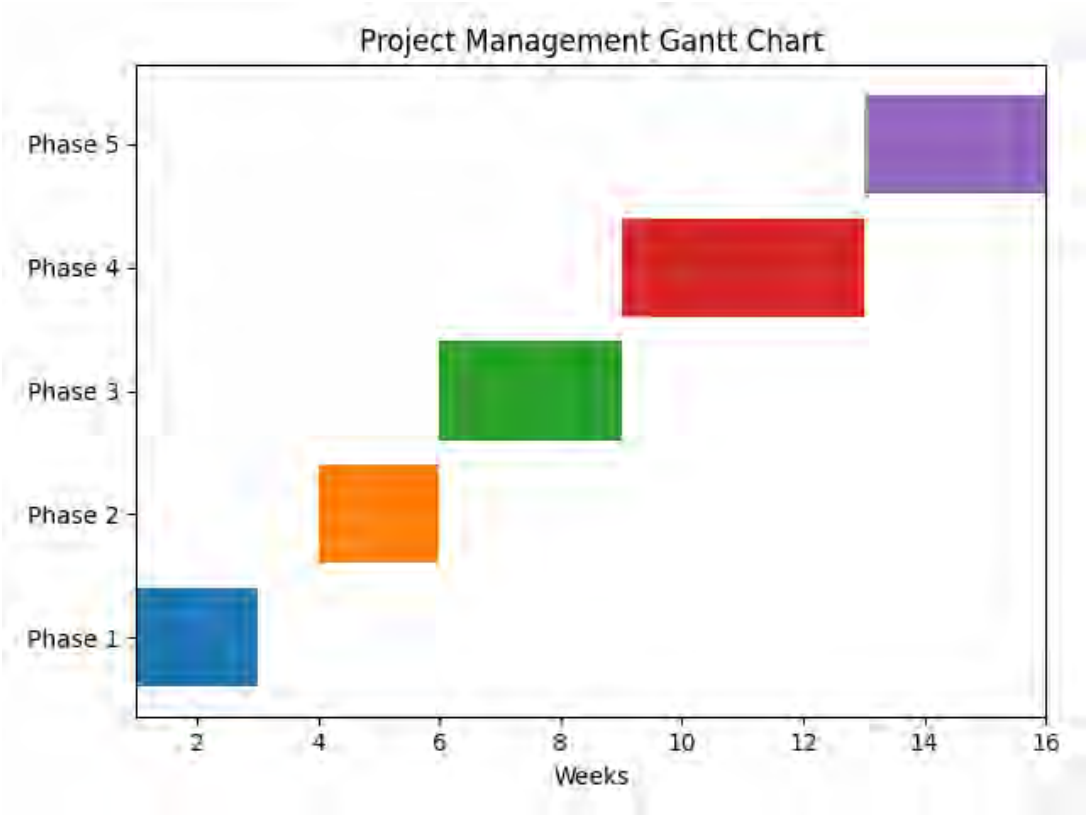


Figure 3.1: Gantt chart of the SPRINT project timeline.

Concurrency within phases helped keep the project on schedule. During Phase 3, the GPU-intensive oracle-labeling runs were executed overnight, freeing daytime hours for router design and debugging. During Phase 4, the pruning engine and reinforcement learning controller were developed in parallel because they interacted through a clean action interface in which the controller emitted an action index and the engine applied the corresponding pruning decision.

Version control and reporting were handled using a single-branch Git workflow supplemented by checkpointed experiment directories. Training reports, ablation outputs, and test results were stored in numbered folders so that each iteration remained traceable over time. This structure improved transparency and reduced the risk of experimental confusion during rapid development.

Two subsystems required substantial mid-project revision. First, the original identity-forward approach for skipping transformer layers caused cache misalignment in Hugging Face's `DynamicCache`, because skipped layers did not update the cache before later layers attempted to read from it. The solution was to physically remove layers from `nn.ModuleList` and then reassign `layer_idx` sequentially. Second, the reward function initially compressed perplexity changes too aggressively, making it difficult for the RL agent to distinguish moderate degradation from severe degradation. Replacing that formulation with a normalized linear perplexity ratio and clamping the result to the range $[-1, 1]$ produced more stable learning behavior.

3.7 Risk Management

Risk management was treated as a continuous activity throughout the project rather than as a one-time planning exercise. Eight major risks were identified at the outset and monitored during implementation, training, and testing. Table 3.5 records each risk, its estimated likelihood and impact, the mitigation strategy adopted, and the eventual outcome.

Table 3.5: Risk register for the SPRINT project.

Risk	Likelihood	Impact	Mitigation	Outcome
LCR router fails to learn a meaningful sensitivity signal	Medium	High	Use a fallback heuristic proxy based on compression ratio and prompt length, alongside iterative architecture refinement	Mitigated: $\rho = 0.797$
RL controller fails to converge to a useful policy	Medium	High	Use random-action ablations as a baseline and sweep reward-function configurations to identify stable learning settings	Mitigated: positive average reward
Pruned inference becomes slower than the dense baseline	High	High	Conduct root-cause analysis on cache misalignment and redesign pruning around physical layer removal	Resolved
GPU out-of-memory occurs during Llama inference	Medium	Medium	Cap sequence length at 128 tokens, set `max_new_tokens=50`, and use single-sample inference without batching	No OOM encountered
Dataset quality problems such as duplicates, noise, or source bias	Low	Medium	Apply automated audit scripts, source-stratified splits, and balanced oversampling during training	10,000 clean prompts obtained
LCR overfits on the relatively small training set	Medium	Medium	Apply dropout, label smoothing, early stopping, weight decay, and differential learning rates	Validation–test gap remained below 0.005 on ρ
Hardware failure or data loss interrupts the project	Low	High	Maintain Git version control and save checkpoints for every best epoch and test iteration	No incidents
Oracle labeling requires excessive iteration time	Medium	Low	Use streaming dataset loading, progress logging with ETA, and a design that supports later parallelization	Approximately 3 hours for 10,000 prompts

The most significant risk that materialized in practice was the possibility that pruned inference would become slower than the dense baseline. During weeks 8 and 9, the initial layer-skipping implementation produced negative speedups even though generated outputs and perplexity values appeared superficially reasonable. The discrepancy became evident only after repeated timing runs showed that the supposedly pruned model was consistently slower than the dense version. Further debugging revealed that skipped layers were not calling `cache.update()`, which caused the dynamic cache to become misaligned and forced later layers to read stale or invalid key-value entries. Once the pruning mechanism was redesigned around physical layer removal and sequential layer-index reassignment, the speedup became both genuine and repeatable.

3.8 Economic Analysis

The proposed framework was developed without a dedicated cloud-computing budget and without any specialized hardware purchases. As shown in Table 3.6, the incremental financial cost of the project was limited almost entirely to electricity consumption, while the hardware, software, models, and datasets were all available beforehand or obtained freely.

Table 3.6: Project cost breakdown for the SPRINT framework.

Category	**Item**	**Cost (USD)**
Hardware (pre-existing)	NVIDIA RTX 5090 GPU (32 GB)	∼2,000
Hardware (pre-existing)	AMD Ryzen 9 9950X + 64 GB DDR5 + 1 TB SSD + 1 TB HDD	∼1,200
Software	Python, PyTorch, and Hugging Face libraries (open-source)	0
Software	Git, VS Code, and Windows license	0
Cloud compute	None used	0
Model weights	Llama-2-7B and BERT-mini checkpoints	0
Datasets	GSM8K, MBPP, WikiText-2, MMLU, and BoolQ	0
Electricity (approximately 80 GPU-hours)	Approximately 575 W × 80 hours ≈ 46 kWh at $0.10/kWh	∼4.60
Total incremental cost		**∼4.60**

The incremental cost of the entire project is therefore under five US dollars in electricity. Since all hardware was pre-existing and all core software components, model checkpoints, and datasets were freely accessible, the project demonstrates that meaningful research on adaptive LLM inference can still be conducted at very low direct cost.

The deployment-time savings are also noteworthy. On the RTX 5090 workstation, the observed average speedup of 32.0% means that a dense Llama-2-7B inference requiring 1,287.61 ms is reduced to approximately 875.39 ms, saving about 412.22 ms per prompt. Because the RTX 5090 has a 575 W thermal design power, these runtime reductions translate directly into meaningful energy savings over sustained use, especially for users who run large prompt volumes locally.

These savings become even more meaningful in sustained local workloads, where reduced GPU active time directly lowers cumulative energy consumption. The framework also avoids recurring cloud API costs, licensing fees, and enterprise hardware requirements. Although the controller introduces its own overhead, the combined routing and action-selection cost averages 17.77 ms, which remains substantially smaller than the latency savings produced by pruning on a dense Llama-2-7B pass. As a result, the adaptive controller effectively pays for itself at inference time while preserving the low-cost deployment profile of the overall framework.

Compared to cloud-based LLM deployment, which incurs recurring API and infrastructure costs, the proposed framework demonstrates a highly cost-efficient alternative by

enabling local inference on consumer hardware.
The minimal operational cost, combined with improved efficiency and reduced hardware requirements, highlights the economic viability of the proposed approach for individual users, small organizations, and research environments.

Chapter 4

Proposed Methodology

4.1 Design Process and Methodology Overview

Static model compression treats every prompt the same way. Quantization [8], knowledge distillation [45], and fixed pruning schedules [9] each produce a single compressed model variant that cannot adapt to what the user is actually asking. A short factual question receives the same heavy processing as a multi-step mathematical derivation. A laptop running on 15% battery applies the same model configuration as a plugged-in workstation with idle GPU cycles. This rigidity wastes compute on easy prompts and risks quality degradation on hard ones.

SPRINT's central hypothesis is that structural pruning decisions should be made per prompt, at inference time, using both a learned estimate of the prompt's sensitivity to pruning and the current hardware state. The research question is direct: can a lightweight router, trained on observed pruning degradation from the actual backbone, guide a reinforcement learning controller to select pruning configurations that achieve measurable speedups without unbounded quality loss?

The answer, based on the experiments reported in Chapter 5, is yes. The system achieves a 32.0% average inference speedup through physical transformer-layer removal while keeping quality degradation bounded and predictable. The learned router generalizes across five diverse benchmark domains with Spearman $\rho = 0.797$ against oracle sensitivity labels.

4.1.1 Research Workflow

We organized the methodology as a multi-stage pipeline where each stage produces artifacts consumed by the next. This sequential dependency is deliberate: it lets each component be developed, validated, and iterated independently before integration. Figure 4.1 illustrates the overall research methodology pipeline.

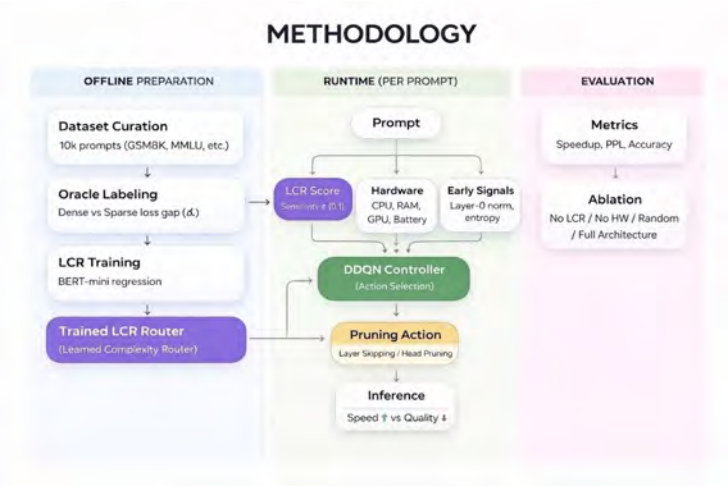


Figure 4.1: Research Methodology Workflow

The pipeline proceeds through five stages:

1. **Dataset curation** assembles a balanced, multi-domain prompt collection from five public benchmarks (GSM8K, MBPP, WikiText-2, MMLU, BoolQ), producing 10,000 prompts with source-stratified train/test splits.

2. **Data preprocessing and feature engineering** cleans the assembled prompts through automated quality filters, aligns tokenizer truncation lengths across all downstream components, and computes nine statistical text features that augment the router's learned representations.

3. **Oracle sensitivity labeling** measures how much each prompt's output quality degrades under specific pruning operations applied to the actual backbone model (Llama-2-7B), assigning each prompt a composite sensitivity score normalized to $[0, 1]$.

4. **The Learned Complexity Router (LCR)**, a fine-tuned BERT-mini encoder [27], learns to predict these oracle scores from the prompt text alone, replacing the expensive oracle with a lightweight routing stage whose end-to-end CPU latency stays under 30 ms.

5. **The reinforcement learning controller**, formulated as a Double DQN [19], integrates the router's sensitivity prediction with hardware telemetry and early backbone signals into a 10-dimensional state vector, then selects from 17 discrete pruning actions using a normalized reward function that balances inference speed against output quality.

Each stage is implemented as a standalone module, and the intermediate artifacts — labeled datasets, model checkpoints, metadata files — provide clear boundaries between components.

4.1.2 Distinction from Prior Work

Runtime adaptive pruning is not new. RAP [5] and similar methods use heuristic prompt-complexity scores or memory-budget controllers to modulate pruning at inference time. SPRINT differs in three specific ways.

First, the router is trained on oracle labels derived from the actual backbone's dense-versus-sparse loss gaps, not on surface-level proxies like token count or keyword density. The label for each prompt directly quantifies how much that prompt degrades under a given pruning configuration of the specific target model, giving it operational meaning that generic difficulty scores lack.

Second, the oracle labels distinguish between pruning methods. Head-pruning sensitivity and layer-skipping sensitivity are only weakly correlated (Spearman $\rho \approx 0.31$). This makes intuitive sense: attention-head pruning disrupts the model's ability to attend to multiple positions simultaneously, while layer skipping removes entire transformer blocks and reduces the model's depth. A prompt that tolerates narrower attention may still suffer from shallower processing, and the reverse also holds. Treating these as interchangeable would discard information that the downstream controller can use.

Third, the layer-skipping engine physically removes layers from the model's internal module list and reassigns layer indices sequentially for correct cache alignment [44]. Prior implementations that monkey-patch forward methods to identity functions cause KV-cache

misalignment: skipped layers do not invoke the cache update routine, so subsequent layers read entries intended for earlier positions. This bug is subtle — the model still generates text — but it causes pruned inference to run slower than the unpruned baseline due to cache thrashing. Physical removal eliminates the problem and produces genuine latency reductions.

4.2 Preliminary Design and Model Specification

This section walks through each component of the SPRINT pipeline in the order it was designed and built. The subsections mirror the pipeline stages: dataset construction, data preprocessing, oracle labeling, the learned router, the RL controller, the pruning engine, and the ablation framework. For each component, we describe what it does, why it is designed the way it is, and how it connects to the stages around it.

Backbone Model Overview

All experiments in this thesis use a single backbone LLM from Meta's Llama family: Llama-2-7B. The choice of backbone determines the pruning action space, the attention architecture, and the hardware requirements.
Llama-2-7B (`meta-llama/Llama-2-7b-hf`) is a 7-billion-parameter decoder-only transformer with 32 transformer layers and 32 attention heads under standard multi-head attention (MHA) [14]. The model uses RoPE positional embeddings and SwiGLU feed-forward activations. In 16-bit precision at 128-token sequence lengths, it requires approximately 14 GB of VRAM, so all experiments were run on an NVIDIA RTX 5090 with 32 GB VRAM. Dataset curation, oracle labeling, LCR training, RL training and testing, and ablation studies all use this same backbone.
Table 4.1 summarizes the selected backbone configuration.

Table 4.1: Backbone model specifications.

Property	**Llama-2-7B**
Parameters	∼6.74 billion
Transformer layers	32
Attention heads	32
KV attention heads	32 (MHA)
Hidden dimension	4,096
FFN intermediate dim	11,008
Positional encoding	RoPE
Attention variant	MHA [20]
VRAM (16-bit, 128 tokens)	∼14 GB
Hardware	RTX 5090 (32 GB VRAM)

4.2.1 Preliminary Design Evaluation

Before settling on the final SPRINT architecture, we evaluated multiple alternative solutions for each major system component. Each candidate was functionally verified through

controlled experiments on a representative subset of the dataset, with results compared against well-defined acceptance criteria derived from the specifications in Chapter 3. This section consolidates those preliminary experiments and documents the engineering rationale behind each design decision. Table 4.2 summarizes the alternatives evaluated and their outcomes.

Table 4.2: Alternative design solutions and functional verification results.

Component	Alternative Evaluated	Verification Method	Outcome	Decision
Router architecture	MLP over bag-of-words features	Trained on the same oracle labels; evaluated with Spearman ρ on the test split	$\rho < 0.45$	Rejected: insufficient capacity to capture contextual sensitivity patterns
Router architecture	DistilBERT (66M params) (Sanh et al., 2019)	CPU inference-latency benchmark	∼60 ms per forward pass	Rejected: substantially slower than BERT-mini's raw forward pass (∼3 ms) and leaves too little end-to-end latency headroom for NFR-4 ($\lesssim$30 ms)
Router architecture	TinyBERT (14.5M params) (Jiao et al., 2020)	CPU inference-latency benchmark	∼12 ms per forward pass	Rejected: materially slower than BERT-mini's raw forward pass without proportional accuracy gain
Router architecture	BERT-mini (11.3M params) [27]	Trained on oracle labels; evaluated on test split	$\rho = 0.797$, ∼3 ms raw CPU forward (∼28 ms end-to-end routing)	Selected: meets latency constraint and exceeds accuracy target ($\rho \geq 0.70$)
Pooling strategy	CLS token embedding	Evaluated regression performance vs. mean pooling	Lower ρ on validation set	Rejected: CLS is pre-trained for next-sentence prediction, not sentence regression [25]
Pooling strategy	Mean pooling over non-padding tokens	Same experiment	Higher ρ on validation set	Selected: incorporates information from all token positions
Oracle label type	Raw perplexity gap ($\text{PPL}_S - \text{PPL}_D$)	Computed label distributions; assessed regression stability	Heavy-tailed: 99% of labels below 0.1 after normalization; training dominated by outliers	Rejected: exponentiated loss creates an unstable target distribution (Jelinek et al., 1977)
Oracle label type	Cross-entropy loss gap ($\ell_S - \ell_D$)	Same dataset; assessed distribution properties	Well-behaved distribution amenable to min-max normalization	Selected: log-space formulation gives unit-free, comparable sensitivity scores
Layer skipping	Identity-forward monkey-patching	Inference speed measured on 100 prompts	Pruned inference 5–15% slower than baseline	Rejected: KV-cache misalignment causes degraded attention and cache thrashing [44]
Layer skipping	Physical removal from module list + index reassignment	Same benchmark	Genuine speedups (up to 50%+ at high intensities)	Selected: correct cache alignment produces real latency reductions
Pruning operators	FFN intermediate-dimension slicing	Trained RL agent with FFN actions included	Consistently negative rewards; reshaping overhead exceeded latency savings	Rejected: poor quality-speed tradeoff; removed from the action space
Pruning operators	Layer skipping + head pruning (no FFN)	Same RL training protocol	Positive rewards; cleaner policy convergence	Selected: both operators produce genuine speed-quality tradeoffs
Reward function	Log-PPL penalty: $R_{\log} = \alpha \cdot \Delta\text{speed} - \beta \cdot \max(0, \ln \text{PPL}_p - \ln \text{PPL}_b)$	RL training over 1,000 episodes	Agent could not distinguish 5× from 50× PPL increase; quality signal over-compressed	Rejected: logarithmic compression removes proportional penalty information
Reward function	Normalized linear PPL ratio with $[-1, 1]$ clamp	Same RL training protocol	Agent learned fine-grained speed-quality tradeoffs; positive converged reward	Selected: preserves proportional relationship between degradation magnitudes

Several observations emerge from this systematic evaluation. For the router, the progression from a surface-level MLP ($\rho < 0.45$) through increasingly capable transformer encoders confirms that contextualized representations are necessary for pruning sensitivity prediction — surface features alone cannot capture the structural and semantic properties that determine how a prompt responds to compression. The BERT-mini sweet spot exists because routing latency is subtracted directly from pruning savings: any router that takes longer than the latency it helps save defeats the framework's purpose. At ∼3 ms, BERT-mini's overhead is negligible against the ∼150–200 ms typically saved through pruning on a 1,300 ms baseline.

The identity-forward bug was the most consequential preliminary finding. The implementation looked correct — the model generated fluent text with plausible perplexity

— but timing measurements revealed that "pruned" inference was consistently slower than the unpruned baseline. The root cause was that skipped layers did not invoke the cache update routine, causing downstream layers to read stale KV-cache entries. This is a correctness failure that would not be caught by output quality checks alone; only systematic speed benchmarking exposed it. The fix (physical removal with sequential index reassignment) turned negative speedups into genuine positive speedups and became one of the framework's three distinguishing contributions over prior work.

The FFN slicing rejection illustrates a practical engineering constraint: not every pruning operator that reduces parameter count produces proportional latency reduction. FFN intermediate dimensions are tightly coupled to the matrix multiplication kernel's efficiency on GPU — reshaping them to non-standard sizes forces the CUDA runtime to use less optimized kernel variants, offsetting the reduction in arithmetic operations. Layer skipping and head pruning do not have this problem because they remove entire structural units without breaking the remaining units' dimensional consistency.

The reward function evolution demonstrates that the RL controller's learning signal must preserve proportional quality information. The log-PPL formulation, while theoretically reasonable as a compression of the heavy-tailed perplexity distribution, over-compressed the signal to the point where the agent could not learn nuanced tradeoffs. The linear formulation with explicit clamping provides both proportionality and boundedness.

These preliminary experiments collectively verified each component's functional correctness and justified each design choice against concrete alternatives, ensuring that the final SPRINT architecture described in the following sections is not an arbitrary configuration but the outcome of systematic engineering evaluation.

Alternative Design Exploration

To ensure robustness and optimal system design, multiple alternative solutions were systematically evaluated across key components of the SPRINT framework. These included variations in router architecture, pruning strategies, reward functions, and layer-skipping implementations.

For example, simpler models such as MLP-based routers were tested but failed to capture contextual sensitivity ($\rho < 0.45$), while larger transformer variants such as DistilBERT introduced unacceptable latency overhead. Similarly, identity-based layer-skipping approaches resulted in degraded performance due to KV-cache misalignment, leading to the adoption of physical layer removal.

These evaluations, summarized in Table 4.2, guided the selection of the final architecture by balancing accuracy, efficiency, and real-world deployment constraints.

4.2.2 Dataset Curation

A runtime pruning system is only as good as the labels that train its router, and those labels are only as good as the prompts they are derived from. Prior work on runtime pruning has relied either on narrow task-specific benchmarks (Dettmers et al., 2022) or on synthetic prompt distributions that do not capture the heterogeneity of real-world LLM usage. We constructed a diverse, multi-domain benchmark mixture drawn exclusively from well-known public datasets, ensuring reproducibility and broad coverage across the principal modalities of LLM workloads. The decision to use established public benchmarks over a custom-collected corpus is deliberate: it enables direct comparison with the

broader literature, avoids distribution-shift concerns that accompany private datasets, and ensures that the sensitivity labels reflect performance on tasks the community has agreed are representative of LLM capabilities.

Source Selection

We selected five benchmark sources to cover five functionally distinct categories of language model prompts. Each source stresses a different axis of model competence, which forces the learned router to generalize across prompt types rather than overfit to a single task distribution. Table 4.3 summarizes our selection and the rationale behind it.

Table 4.3: Benchmark sources and selection rationale.

Source	Count	Domain	Rationale
GSM8K	2,000	Mathematical reasoning	Grade-school math word problems with multi-step arithmetic and logical chains [17]. Intermediate errors compound under pruning, making math prompts highly sensitive to structural compression.
MBPP	2,000	Code generation	Python programming tasks written in natural language (Austin et al., 2021). Balanced brackets, indentation, and variable scope create long-range dependencies that are sensitive to attention-head removal.
WikiText-2	2,000	Language modeling	Long-form Wikipedia paragraphs used as a canonical language-modeling benchmark [42]. High lexical redundancy and predictable structure make narrative text relatively robust to moderate pruning, providing a counterweight to more sensitive sources.
MMLU	2,000	Mixed-domain knowledge	Multiple-choice questions spanning 57 academic subjects [16]. The structured format provides clear answer boundaries, while the subject breadth tests domain generalization under pruning.
BoolQ	2,000	Reading comprehension	Passage-grounded yes/no questions from Google search queries (Clark et al., 2019). The passage+question format stresses reading comprehension and passage-question alignment under structural compression.

The equal contribution of 2,000 prompts per source yields a balanced 10,000-prompt mixture. This balance matters because source-imbalanced datasets bias the router toward over-representing the dominant source's sensitivity distribution, leading to poor generalization on minority sources — a well-documented concern in multi-task learning (Raffel et al., 2020).

Construction Pipeline

The dataset construction proceeds through three automated stages.
In the first stage, prompts are streamed from the Hugging Face Datasets API (Lhoest et al., 2021) to avoid downloading entire datasets to disk. Each source is loaded from its canonical split and formatted into a uniform prompt string. GSM8K uses the raw question field. MBPP uses the natural-language instruction field. WikiText-2 paragraphs are filtered to remove section headings, short fragments (fewer than 50 characters), and low-word-count lines (fewer than 8 words), retaining only substantive prose. MMLU questions are formatted with labeled answer choices (A, B, C, D) appended to the question stem, matching the standard evaluation format. BoolQ concatenates the passage and question into a structured format with binary answer options.
Every row carries metadata columns for downstream stratification: the originating dataset, the split within that dataset, a unique identifier, subject category, and context depen-

dency type. An 80/20 stratified split column is appended for reproducible train/test partitioning, ensuring each benchmark source maintains proportional representation in both splits (Hastie et al., 2009).
In the second stage, the assembled dataset undergoes automated quality filtering. Exact-duplicate detection uses hash-based deduplication over the full prompt text. Empty or whitespace-only prompts are discarded. A minimum token-length threshold removes excessively short prompts. Malformed rows with missing required columns are caught and removed. Per-source audit reports document exactly which rows were removed, by which filter, and how many.
The final stage produces the canonical 10,000-prompt dataset consumed by all downstream components.

Why Raw Prompt Text

We deliberately avoided preprocessing the prompt text beyond source-specific formatting and quality filtering. Pruning sensitivity is a property of how the backbone LLM actually processes the prompt, including all formatting choices, punctuation, and casing decisions made by the source datasets. Normalizing text — lowercasing, removing punctuation, stripping whitespace — would destroy structural information that affects attention patterns and layer-specific processing. Each source dataset has a canonical format established in its original paper [17, 16, 42], together with Austin et al. (2021) and Clark et al. (2019), and respecting these formats ensures that sensitivity labels reflect genuine task characteristics rather than artifacts of our preprocessing.
At inference time, prompts are tokenized and truncated identically to how the oracle processed them (128 tokens), so there is no train-test distribution mismatch. This alignment follows best practices in transfer learning where input processing must be identical between training and inference [15].

4.2.3 Data Preprocessing and Feature Engineering

Prompts in the assembled dataset pass through a multi-layer preprocessing pipeline before any model training begins. This pipeline serves three functions: removing low-quality text that would introduce noise into oracle labels, aligning representational boundaries across the oracle, router, and RL components, and computing statistical text features that provide explicit domain signals to the router.

Quality Filtering

The quality audit applies a sequence of filters independently to each benchmark source, with source-specific thresholds calibrated to the typical length distributions of each domain.
For most sources, a minimum character threshold of 30 characters is enforced. WikiText-2 uses a stricter 50-character threshold because Wikipedia prose tends to have longer substantive sentences, and very short fragments in WikiText are usually heading artifacts or list entries rather than continuous prose. A minimum word count (5 words for most sources, 8 for WikiText-2) further removes telegraphic one-liner entries. A low-alpha-content check removes rows where the fraction of alphabetic characters falls below a structural threshold — this catches metadata rows, URL-heavy entries, and formatting

artifacts that slipped through the regex-based heading filter. Finally, any empty or whitespace-only prompt is discarded entirely.
After all filters, a final deduplication pass removes any exact-text duplicates that may have been introduced through the streaming assembly process (e.g., if a question appears in both the training and test split of a source dataset). Per-source audit reports are generated documenting the count of rows removed by each filter, enabling transparent inspection of data quality decisions.

Tokenizer-Level Truncation

Both the oracle labeling pass and the LCR training pipeline truncate prompt text to exactly 128 tokens using their respective tokenizers (the Llama SentencePiece tokenizer for the oracle, the WordPiece tokenizer for the BERT-mini router). This shared truncation length is a deliberate design constraint, not a capacity limit.
If the oracle evaluated the first 128 tokens of a prompt while the LCR processed all 512 tokens, the LCR would base its sensitivity prediction on content that the oracle's label never evaluated. The resulting distribution mismatch would systematically degrade router accuracy: the LCR would learn to predict sensitivity scores derived from 128-token representations but would see different statistical properties in the 512-token prompts at inference time. The 128-token constraint eliminates this misalignment entirely and is consistent with the train-test alignment principle established in the transfer learning literature [15].
Practically, as noted above, 128 tokens is sufficient to capture the structural identity of prompts across all five benchmark sources. The truncation primarily affects a subset of longer WikiText passages.

Auxiliary Feature Engineering

Nine lightweight statistical features are computed from the raw prompt text at both training time (for LCR fine-tuning) and inference time (for the SPRINT runtime pipeline). These features are computed independently of any neural model and are available in under 0.1 ms each on CPU.
The features are designed to be explicitly complementary to the BERT-mini contextualized embeddings, not redundant with them. The BERT embedding captures latent semantic and syntactic content through learned attention patterns; the auxiliary features capture statistical structure and domain signals that BERT may not surface without specific fine-tuning. The combination of learned representations with domain-specific hand-crafted features is a well-established hybrid approach in NLP feature engineering (Zhou et al., 2015).
Table 4.4 lists all nine features with their computation and justification.

Table 4.4: Auxiliary text features for the LCR router.

Feature	Computation	Normalized Range	Rationale
Log token count	$\log(1 + \text{word count})$	$\approx [0, 1]$	Captures prompt length on a compressed scale; longer prompts provide more context but also more content that can degrade under pruning
Compression ratio	len(zlib(text))/len(text in bytes)	$[0, 1]$	Proxy for information entropy [40]; low ratio indicates repetitive text (nature prose); high ratio indicates dense, varied content (code, math)
Avg word length	Mean character count per word / 12	$\approx [0, 1]$	Longer words suggest technical or domain-specific vocabulary; useful for separating MBPP and GSM8K from WikiText
Special char ratio	count(non-alphanumeric)/total chars	$[0, 1]$	Elevated in code and mathematical notation; helps identify MBPP-type prompts
Unique token ratio	|unique words|/|total words|	$[0, 1]$	Lexical diversity indicator; low for repetitive narrative prose, high for varied technical content
Code markers	Binary: presence of `def`, `class`, `import`, {, }	$\{0, 1\}$	Hard domain signal for Python code generation prompts (MBPP)
Numeric density	count(digit tokens)/total tokens	$[0, 1]$	High in mathematical reasoning prompts (GSM8K); correlates with sensitivity to arithmetic errors under pruning
Question detection	Binary: presence of `?` or question words	$\{0, 1\}$	Flags question-answering format prompts (BoolQ, MMLU)
Avg sentence length	Mean word count per sentence / 40	$\approx [0, 1]$	Longer sentences suggest complex syntactic structures; proxy for sentence-level complexity

All nine features are concatenated with the 48-dimensional attention statistics vector (described in Section 4.2.5) and projected through a single linear layer (GELU activation) from 57 to 48 dimensions, producing the auxiliary embedding that feeds into the router's regressor head.

Training-Time Augmentation

Two augmentation strategies are applied during LCR training to improve generalization on the 7,200-sample training set.

Source-balanced oversampling duplicates samples from minority sources (with replacement) until all five sources contribute equally to each training epoch. Without balancing, the model's parameter updates are dominated by sources with larger sample counts, developing weaker representations for underrepresented domains. This follows the imbalance correction literature [41] adapted for a regression setting.

Label smoothing adds uniform noise $\mathcal{U}(-0.01, +0.01)$ to oracle labels during training, clamped to $[0, 1]$. As discussed in Section 4.2.4, oracle labels carry inherent measurement noise from stochastic GPU computation. The ± 0.01 perturbation explicitly teaches the model that the labels are soft targets with uncertainty, preventing overconfident fitting to the precise label values [35, 34].

4.2.4 Oracle Sensitivity Labeling

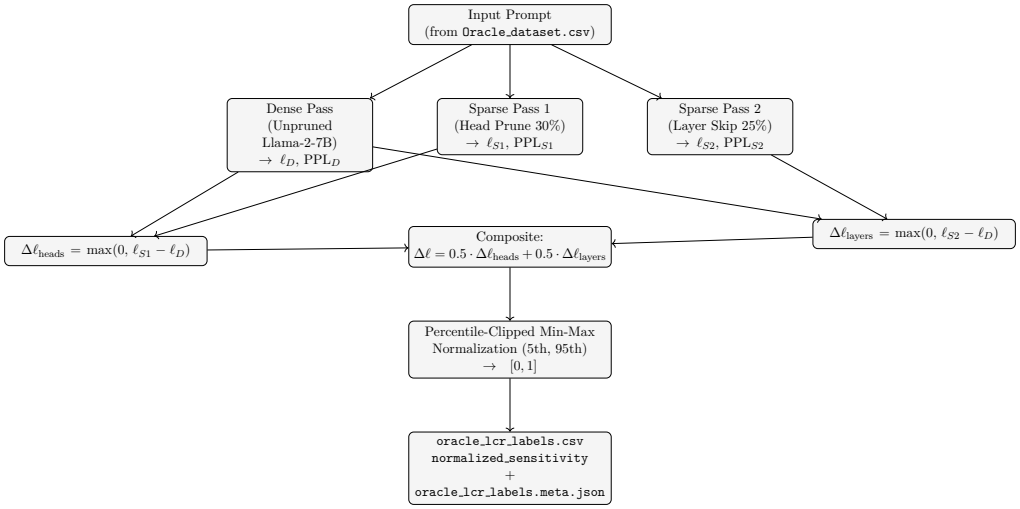


Figure 4.2: Oracle sensitivity labeling pipeline

The oracle labeling stage produces the ground-truth target variable for the router. Figure 4.2 summarizes the full oracle sensitivity labeling pipeline, from dense and sparse inference passes through loss-gap composition and normalization. Unlike generic "prompt difficulty" scores based on surface features — token count, perplexity, or lexical complexity (Ethayarajh, 2019) — oracle labels are observed degradation of the actual backbone under specified pruning operations. A high sensitivity score means the prompt's output quality degrades substantially when the model is pruned; a low score means the prompt is robust to structural compression. This distinction from surface-level difficulty is critical: a long prompt is not necessarily pruning-sensitive, and a short prompt is not necessarily pruning-robust. The oracle measures what actually happens when the model is compressed.

Labeling Protocol

For each prompt, the oracle pipeline runs two types of inference pass on the Llama-2-7B backbone.
A dense teacher-forcing pass processes the prompt through the unpruned model using next-token prediction with ground-truth input tokens, producing a dense cross-entropy loss ℓ_D and dense perplexity $\text{PPL}_D = \exp(\ell_D)$. Then, for each pruning configuration, the backbone is structurally pruned, the same prompt is processed, and a sparse loss ℓ_S and sparse perplexity PPL_S are recorded. After each sparse pass, the model is fully restored to its original dense state, preventing information leakage between sparse configurations. The principal label is the non-negative loss gap:

$$\Delta\ell = \max(0,\ \ell_S - \ell_D) \tag{4.1}$$

We chose the loss gap over a raw perplexity gap ($\text{PPL}_S - \text{PPL}_D$) for two reasons grounded in the statistical properties of language model evaluation (Jelinek et al., 1977; Manning & Schütze, 1999). First, perplexity is an exponentiated loss. A small loss difference (e.g., $\Delta\ell = 2$) maps to a perplexity ratio of $e^2 \approx 7.4$, but a large difference ($\Delta\ell = 8$) maps to

$e^8 \approx 2981$. Working in log-space avoids this heavy-tailed instability that would dominate regression training and skew the learned sensitivity distribution. Second, the loss gap corresponds to a log-perplexity ratio ($\Delta\ell = \ln(\text{PPL}_S) - \ln(\text{PPL}_D)$), making it directly comparable across prompts with different baseline perplexities — a unit-free measure of relative degradation.

The non-negativity constraint ($\max(0, \cdot)$) clamps the rare cases where pruning accidentally improves the loss (through noise or regularization effects), ensuring that sensitivity labels are always ≥ 0 and that the downstream regression target has a well-defined lower bound.

Multi-Method Composite Labels

We use two pruning configurations simultaneously to produce composite sensitivity labels:

- Attention-head pruning at 30% intensity (removing approximately 10 of 32 attention heads).
- Transformer-layer skipping at 25% intensity (removing 8 of 32 layers).

Each prompt receives per-method loss gaps $\Delta\ell_{\text{heads}}$ and $\Delta\ell_{\text{layers}}$, combined into a composite score:

$$\Delta\ell_{\text{composite}} = \frac{1}{2}\Delta\ell_{\text{heads}} + \frac{1}{2}\Delta\ell_{\text{layers}} \tag{4.2}$$

The rationale for multi-method compositing comes directly from our empirical data. The Spearman correlation between head-pruning gaps and layer-skipping gaps across 10,000 prompts is only $\rho \approx 0.31$ ($p < 0.001$, $R^2 \approx 0.05$). Head-pruning sensitivity explains only about 5% of the variance in layer-skipping sensitivity — they are nearly independent signals. This is expected: head pruning removes attention computation capacity within each layer, while layer skipping removes entire transformer blocks. These operations stress fundamentally different model components: attention-pattern formation versus progressive feature refinement through the network's depth. A composite label captures both modes of degradation, giving the downstream router a complete picture of each prompt's vulnerability.

Training on the composite teaches the model: if a prompt is sensitive to either pruning type, predict high sensitivity. This is a conservative strategy — the cost of over-pruning a sensitive prompt (catastrophic quality loss) is worse than the cost of under-pruning a robust one (leaving some speed on the table).

Normalization

Raw composite gaps are normalized into the interval $[0, 1]$ using percentile-clipped min-max scaling:

$$y = \frac{\text{clip}(\Delta\ell, q_5, q_{95}) - q_5}{q_{95} - q_5} \tag{4.3}$$

Percentile clipping at the 5th and 95th boundaries prevents extreme outliers from dominating the normalization range. Without it, a single prompt with an anomalously high loss gap could stretch the range so that 99% of labels fall below 0.1, severely reducing the effective resolution of the target variable. This is a standard robustness technique in feature engineering for regression targets (Hastie et al., 2009). The clipping thresholds, normalization bounds, backbone model identifier, sparse configurations, sequence length, and runtime statistics are recorded in a JSON sidecar metadata file for full reproducibility.

Truncation Alignment

Oracle truncation is set at 128 tokens, matching the router's maximum sequence length. This alignment is critical: if the oracle evaluated prompts at one truncation length while the router processed them at another, the resulting distribution mismatch would degrade accuracy. A prompt truncated to 256 tokens might exhibit different sensitivity characteristics than the same prompt truncated to 128 tokens, because additional context can change attention patterns and pruning sensitivity. This alignment principle follows best practices in transfer learning [15].

Practically, 128 tokens is sufficient to identify the prompt's domain and structural properties. GSM8K problem statements average approximately 120 tokens; MBPP specifications are typically under 80; WikiText passages are truncated to their first 128 tokens; BoolQ passages with questions average roughly 100; MMLU questions with answer choices average about 80. The limit only truncates some longer WikiText passages, and for those, the first 128 tokens reliably identify the domain and structure.

The oracle pipeline is the most computationally expensive stage of dataset preparation (approximately 3 dense-equivalent forward passes per prompt $\times$ 10,000 prompts), but it runs only once. The resulting labels are reusable across multiple router variants without re-running inference.

4.2.5 Learned Complexity Router (LCR)

The Learned Complexity Router is the methodological centerpiece that distinguishes SPRINT from prior runtime-pruning systems relying on heuristic prompt-complexity proxies. It replaces hand-crafted equations — based on token count, regex-matched keyword density, and raw perplexity thresholds — with a learned function that maps from prompt text to a continuous sensitivity score in $[0, 1]$.

Backbone Selection and Justification

We chose `prajjwal1/bert-mini` [27] as the LCR backbone. It is one of Google Research's compact BERT variants, pretrained on the same data as BERT-base but with a reduced architecture, and Turc et al. [27] showed that such models retain strong downstream performance.

Three requirements drove this choice. First, ultra-low latency: the full routing stage had to remain below 30 ms on CPU. BERT-mini has 4 layers, 256 hidden dimensions, 4 attention heads, and 11.3M parameters. Its raw forward pass takes about 3 ms on CPU and under 1 ms on GPU, while the full routing path — including tokenization, feature assembly, and post-processing — takes about 28 ms on CPU. BERT-base, by contrast, has 110M parameters (roughly 10$\times$ more) and would add about 150 ms per pass, erasing much of the pruning speedup.

Second, architectural fit: prompt sensitivity is a whole-sequence regression problem, not a generative one. Bidirectional encoders like BERT produce richer sequence representations than autoregressive decoders for classification and regression because they attend to both left and right context [15]. Small BERT variants have also been shown to outperform similarly sized decoder-only models on efficient text classification tasks [33].

Third, sufficient capacity within bounded complexity: BERT-mini still provides 256-dimensional contextual representations, expanded to 304 dimensions after auxiliary features, which is adequate for predicting a target in $[0, 1]$. Alternatives such as TinyBERT

and DistilBERT were 2–4× larger with no proportional gain on this regression task, while a bag-of-words MLP achieved only Spearman $\rho < 0.45$. Similar compact BERT choices are also supported by later work on efficient NLI systems [28].

Architecture

The LCR consists of four learnable components that together transform a raw prompt string into a scalar sensitivity prediction. Figure 4.3 illustrates the complete architecture.

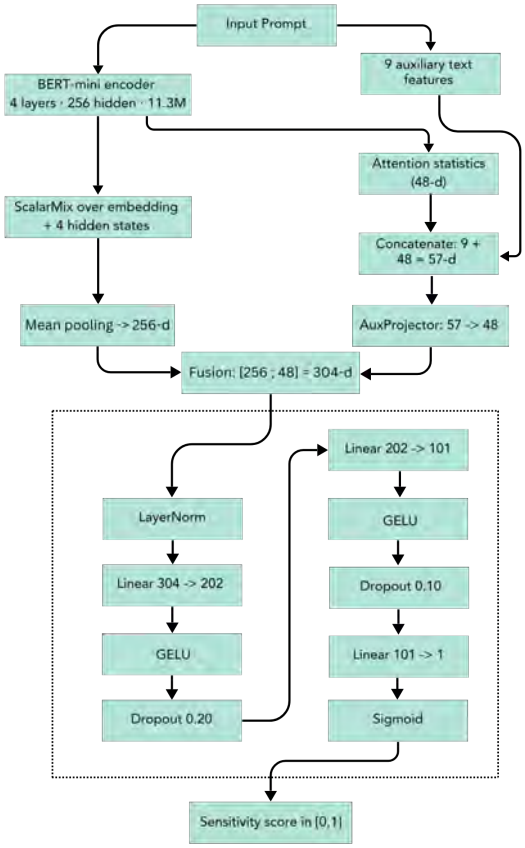


Figure 4.3: LCR architecture diagram.

ScalarMix (Multi-Layer Weighted Pooling)
Rather than using only the final hidden layer of BERT — which captures primarily semantic features — we employ ScalarMix, a mechanism introduced by Peters et al. [22] in the ELMo paper. ScalarMix computes a learned weighted sum over all hidden states (embedding layer + 4 encoder layers), allowing the model to attend to lexical (early layers), syntactic (middle layers), and semantic (late layers) representations simultaneously.
This design is motivated by research from Tenney, Das, and Pavlick [23], who demonstrated through probing classifiers that BERT's different layers encode qualitatively different types of linguistic information: lower layers capture surface and morphological features, middle layers syntactic features, and higher layers semantic and task-specific features. For pruning sensitivity prediction, we need all three levels. A prompt with complex syntax (nested code) may be pruning-sensitive for syntactic reasons, while a

prompt with rare vocabulary (specialized medical terms) may be sensitive for lexical reasons. The learned weights w_i pass through softmax, and the result is scaled by a learned parameter γ:

$$\text{ScalarMix}(h_0, \ldots, h_4) = \gamma \sum_{i=0}^{4} \text{softmax}(w)_i \cdot h_i \tag{4.4}$$

The cost of ScalarMix is just 6 learned parameters (5 weights + 1 scalar). After ScalarMix, mean pooling produces a 256-dimensional sentence embedding. We use mean pooling over all non-padding token embeddings rather than the CLS token, following Reimers and Gurevych [25], who demonstrated in the Sentence-BERT paper that mean pooling consistently outperforms CLS pooling for sentence-level regression tasks. The CLS token is pretrained to aggregate information for next-sentence prediction; mean pooling explicitly incorporates information from every token position, which is mechanistically correct for a task that depends on properties distributed throughout the prompt text.

Attention Statistics Extractor

Clark et al. [24] demonstrated that BERT's internal attention patterns encode meaningful linguistic structure: syntactic dependencies, coreference, and positional patterns. Building on this finding, we extract two statistics from each BERT layer and attention head: attention entropy (how diffusely the attention is distributed across tokens) and maximum attention weight (how concentrated the attention is on a single token). A head with high entropy attends broadly, characteristic of prompts with global syntactic ambiguity. A head with low entropy attends to specific tokens, characteristic of structured code or factual recall.

These are model-internal signals unavailable from surface text features alone. A compressed code snippet and a compressed mathematical statement might have similar lexical density, but their structural complexity — and thus pruning sensitivity — differs. The attention statistics capture how the encoder internally responds to the prompt's structural complexity. The raw features (4 layers $\times$ 4 heads $\times$ 2 stats = 32 scalars) are concatenated with a 16-dimensional learned linear projection (Tanh activation), yielding 48 attention-derived features. The projection layer enables gradient flow back through the attention extractor into the BERT backbone during fine-tuning, creating a bidirectional coupling between feature extractor and encoder — the backbone learns to produce more informative attention patterns for our task.

Auxiliary Text Features

Nine lightweight statistical features, each costing under 0.1 ms to compute, provide explicit domain signals that complement the contextualized representations. These features — described in full in Table 4.4 in Section 4.2.3 — encode statistical structure and domain identity signals that the BERT embedding does not surface without task-specific fine-tuning: prompt length (log-scaled), zlib compression ratio as an information-entropy proxy [40], word length, special-character density, lexical diversity, code-marker detection, numeric density, question-format detection, and average sentence length. The nine auxiliary scalars are concatenated with the 48 attention-statistics features and projected through a linear layer (GELU activation) from 57 dimensions to 48, producing the auxiliary embedding.

Regressor Head

The 256-dimensional mean-pooled BERT representation is concatenated with the 48-dimensional auxiliary embedding, producing a 304-dimensional fused vector. This passes

through a three-layer regression head:

$$304 \xrightarrow{\text{LayerNorm}} 202 \xrightarrow{\text{GELU, Dropout(0.2)}} 101 \xrightarrow{\text{GELU, Dropout(0.2)}} 1 \xrightarrow{\sigma} (0, 1) \tag{4.5}$$

The hidden dimensions follow a progressive 50% reduction pattern standard for regression heads in fine-tuned transformer models. LayerNorm at entry normalizes the concatenated vector, preventing scale mismatch between the BERT embeddings and auxiliary features from dominating initial gradients. We use GELU activations [21] throughout for consistency with the BERT encoder's internal activations. Unlike ReLU, which hard-clips at zero, GELU has a smooth non-zero gradient for negative inputs, which helps gradient flow through the deeper layers. The Sigmoid output squashes the prediction to the unit interval, matching the normalization range of oracle labels. Dropout at 0.20 provides regularization against overfitting on the ∼7,200 training samples.
If the trained checkpoint is absent at runtime, the system falls back to a heuristic proxy based on compression ratio and prompt length. This fallback exists solely for operational robustness; it is not part of the reported contribution.
Figure 4.4 presents lCR training and test-evaluation pipeline.

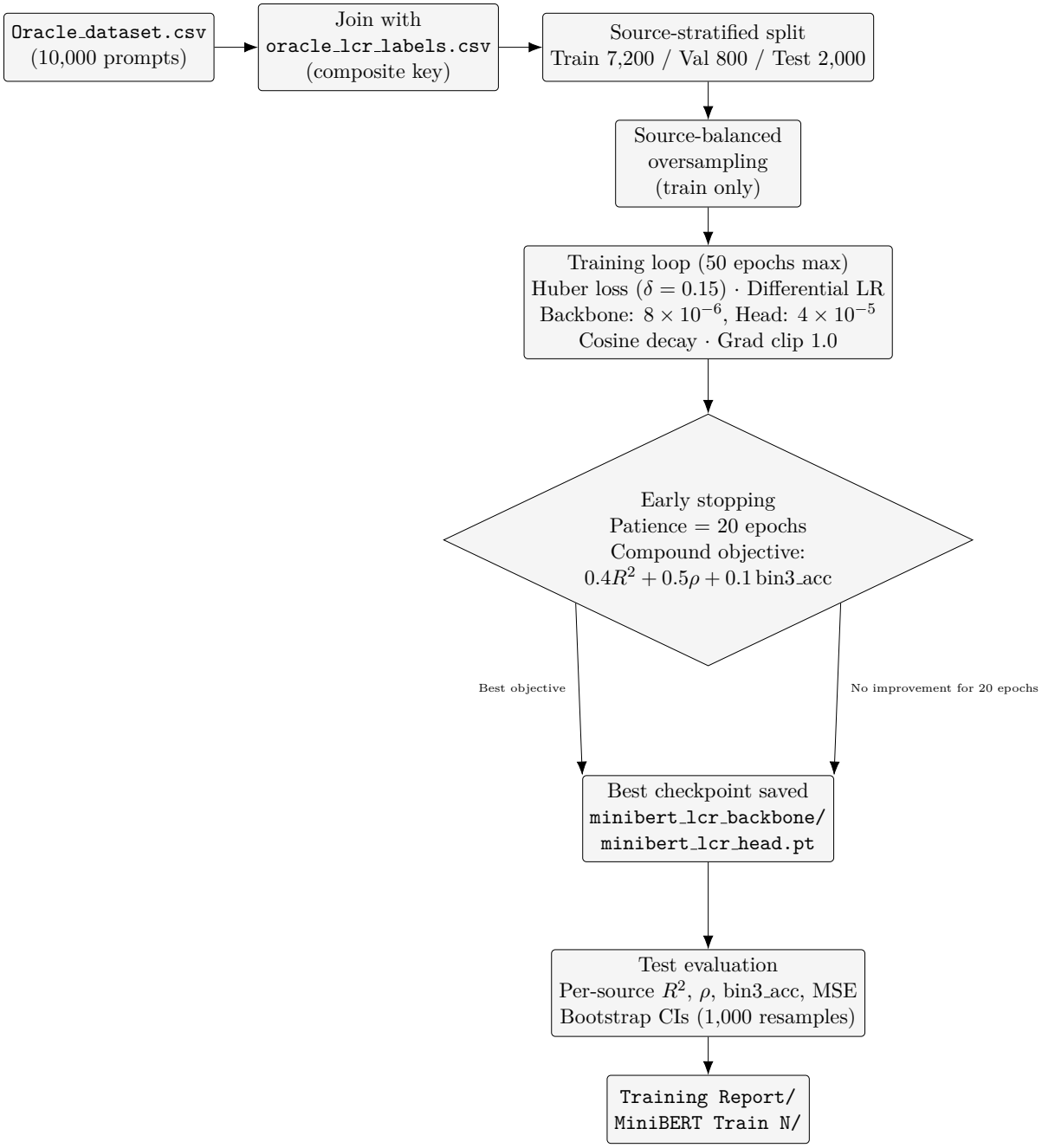


Figure 4.4: LCR training and test-evaluation pipeline.

Training Protocol

The training pipeline follows a rigorous protocol designed to maximize diagnostic value while preventing information leakage between train and test splits.

Data Loading and Split Strategy

The training module supports a two-file workflow: the prompt dataset and the oracle labels reside in separate files joined on a composite key at load time. This separation is a deliberate architectural choice: the prompt dataset remains a clean, reusable artifact consumed by multiple downstream components (RL training, ablation studies), and label regeneration (with different pruning configurations or normalization schemes) does not require re-formatting prompts.

Within the pre-existing 80/20 train/test split, a further 10% of train rows are held out for validation and early stopping. The resulting split is source-stratified, ensuring proportional representation:
Table 4.5 presents data split distribution.

Table 4.5: Data split distribution.

Split	**GSM8K**	**MBPP**	**WikiText**	**MMLU**	**BoolQ**	**Total**
Train	1,440	1,440	1,440	1,440	1,440	7,200
Val	160	160	160	160	160	800
Test	400	400	400	400	400	2,000

Training Loop
Each epoch proceeds through carefully designed steps. *Source-balanced oversampling.* All five sources are oversampled (with replacement) to the count of the largest source within the train split, so each source contributes equally per epoch. This prevents the optimizer from overfitting to whichever source has the widest sensitivity distribution. Without balancing, the model would see some domains less frequently, developing weak representations for their sensitivity signals. The benefit is directly visible in our results: the final checkpoint achieves Spearman $\rho = 0.892$ on MBPP, its highest per-source metric, whereas earlier runs without source balancing showed code-generation prompts as consistently the weakest domain. The approach follows Chawla et al. [41] for handling class imbalance in regression settings.
Label smoothing. Training labels are perturbed by uniform noise $\mathcal{U}(-0.01, +0.01)$ and clamped to $[0, 1]$. This acts as output regularization that prevents the model from becoming overconfident in its predictions near label boundaries [34]. Müller, Kornblith, and Hinton [35] provided the theoretical justification: label smoothing prevents models from overfitting to exact label values and encourages predictions that generalize. Our oracle labels are measurements from stochastic GPU computation — running Llama-2-7B on the same prompt twice may produce slightly different loss values due to floating-point non-determinism and GPU kernel scheduling. The ± 0.01 smoothing is calibrated to this expected measurement noise, teaching the model that exact label values carry inherent uncertainty.
Loss function. We use the Huber loss [31] with $\delta = 0.15$. The Huber loss is the standard choice for regression problems where target labels contain noise or outliers, widely adopted in object detection for bounding box regression [32] precisely because of its robustness properties. In our case, approximately 5% of prompts produce anomalous oracle outputs — very high perplexity spikes or near-zero losses — due to the backbone's intrinsic variability on edge-case inputs. MSE would assign quadratic penalties to these outlier residuals, dominating the gradient and pulling the model toward outliers rather than the mean behavior. Huber loss mitigates this by switching to linear (MAE) penalty for residuals above δ. With $\delta = 0.15$, residuals below 15% of the normalized label range are treated as normal and penalized quadratically, providing precise gradients; residuals above that threshold are penalized linearly, bounding their influence on training. The threshold corresponds to the typical noise floor in our oracle measurements (GPU inference has approximately 5–15% stochastic variability depending on CUDA kernel scheduling).

Differential learning rates. The backbone parameters receive a learning rate of $0.2 \times 4 \times 10^{-5} = 8 \times 10^{-6}$, while the head, projector, ScalarMix, and attention extractor parameters receive the full 4×10^{-5}. This follows the discriminative fine-tuning principle introduced by Howard and Ruder [26], where different learning rates are applied to different layers of a pre-trained model. The BERT-mini backbone has been pre-trained on a large text corpus and encodes valuable linguistic representations. Fine-tuning it at the same rate as randomly initialized components risks catastrophic forgetting — the pre-trained representations get overwritten by task-specific signals before the head has stabilized. The $0.20\times$ factor was chosen empirically from the range commonly used in BERT fine-tuning (typical values are 0.05–0.30 of the head learning rate), following the recommendations of Sun et al. [33], who found that smaller backbone learning rates relative to classification heads produce better and more stable fine-tuning outcomes.

Cosine decay scheduling. After a linear warmup phase (15% of total steps), the learning rate decays following a cosine schedule [29]. The warmup addresses a training stability problem specific to transformers: at the start of training, the ScalarMix weights are uniform, the auxiliary projector is randomly initialized, and the attention extractor is random. With a full learning rate from step one, gradients are large and unstable. Linear warmup ramps the learning rate from near zero to the peak over the first 15% of steps, by which point the randomly initialized components have begun to align on the training signal. The cosine decay after warmup is particularly beneficial for fine-tuning because it decays aggressively early on (when the model is still moving through the loss landscape) and very slowly near the end (when fine-grained adjustments matter most). Devlin et al. [15] established this warmup-plus-decay pattern as standard for all BERT fine-tuning.

Gradient clipping. Gradients are clipped to a maximum L2 norm of 1.0 to prevent training instability from outlier batches.

Validation, Early Stopping, and Model Selection

After each epoch, the model is evaluated on the held-out validation set. A compound objective is computed:

$$\text{Objective} = 0.4 \times R^2 + 0.5 \times \rho + 0.1 \times \text{bin3\_acc} \tag{4.6}$$

The weights reflect downstream importance. Spearman rank correlation ρ receives the highest weight (0.5) because the RL controller cares about relative ranking of prompts more than absolute calibration — if the router correctly identifies that prompt A is more sensitive than prompt B, the controller can assign more conservative pruning to A regardless of the exact predicted value. In the learning-to-rank literature, this distinction between regression accuracy and ranking quality is well established: a model with low MSE that predicts near-constant values would be useless because it provides no ranking information. R^2 (weight 0.4) measures the proportion of variance in oracle labels explained by predictions, providing a calibration-sensitive complement. Three-bin classification accuracy (weight 0.1) is an operational sanity check — prompts binned into low ($[0, 0.33]$), medium $(0.33, 0.67]$, and high $(0.67, 1.0]$) sensitivity should be correctly separated.

The best model by this compound objective is saved, and training terminates if no improvement is observed for 20 consecutive epochs (patience-based early stopping).

Test Evaluation

After training completes, the best-epoch checkpoint is evaluated on the held-out 2,000-prompt test set. Per-source metrics (R^2, Spearman ρ, MSE, MAE, 3-bin accuracy)

diagnose whether the router generalizes uniformly across benchmark types. Bootstrap confidence intervals (1,000 resamples) are computed for the test Spearman ρ to provide statistical precision bounds.
Table 4.6 presents lCR hyperparameter summary.

Table 4.6: LCR hyperparameter summary.

Parameter	**Value**	**Reference / Justification**
Backbone	BERT-mini (4L, 256H, 4A)	[27]; ∼3 ms raw CPU forward, <1 ms GPU, ∼28 ms end-to-end CPU routing
Max sequence length	128 tokens	Matches oracle truncation; [15] alignment principle
Fused input dimension	304	256 (BERT mean-pool) + 48 (auxiliary + attention projection)
Dropout	0.20	Standard for small fine-tuned models
Batch size	48	Fits GPU memory with BERT-mini
Epochs	50 max (early stop, patience 20)	Prevents overfitting on ∼7,200 samples
Learning rate	4×10^{-5}	[15] recommended range for BERT
Backbone LR factor	0.20×	[26]; prevents catastrophic forgetting
Optimizer	AdamW	[30]; correct weight decay decoupling
Weight decay	0.03	[30]; L2 regularization
Label smoothing	±0.01	[34]; [35]
Loss function	Huber ($\delta = 0.15$)	[31]; robust to oracle label outliers
Warmup ratio	0.15	[15]; [29]
Gradient clip norm	1.0	Prevents instability from outlier batches
Pooling strategy	Mean pooling	[25]; superior for regression
Activation function	GELU	[21]; smooth gradient flow

The R^2 Noise Ceiling

The final LCR achieves a test $R^2 = 0.633$, meaning it explains approximately 63% of the variance in oracle sensitivity labels. A panel might interpret the remaining 37% as a model deficiency, but it is more accurately understood as a noise ceiling — a fundamental upper bound on what any model predicts from prompt text alone.
Oracle sensitivity labels are computed from running the full Llama-2-7B backbone under specific pruning configurations. The sensitivity of a prompt is partly determined by factors that are intrinsic to the backbone's internal state — specific attention head acti-

vations, residual stream magnitudes, and layer-specific gating patterns — that cannot be observed from the prompt text without running the model. No text-only predictor can recover these backbone-internal dynamics without access to an internal forward pass. The empirical $R^2 \approx 0.63$ means that approximately 63% of sensitivity variance is indeed predictable from the prompt's surface and contextual properties; the remaining 37% arises from backbone-internal dynamics not observable from text alone. This is consistent with similar noise ceilings observed in model behavior prediction tasks (Ethayarajh, 2019), where the gap between human annotation and model output creates an upper bound for any supervised predictor.

4.2.6 Reinforcement Learning Controller

The RL controller is the decision-making core of SPRINT. It observes a multimodal state representation, combining hardware telemetry, the learned prompt sensitivity score, and early backbone signals, then selects a structural pruning action before inference begins. We formulated it as a Double Deep Q-Network (DDQN) [19], which addresses the well-known Q-value overestimation bias of standard DQN [18] by decoupling action selection (performed by the policy network) from target evaluation (performed by the periodically updated target network). This decoupling matters in our setting because the reward distribution is multi-modal: some actions produce consistently positive rewards while others produce extreme negative rewards, and overestimation of Q-values for high-risk actions would lead to catastrophic policy choices.
Figure 4.5 presents dDQN controller architecture.

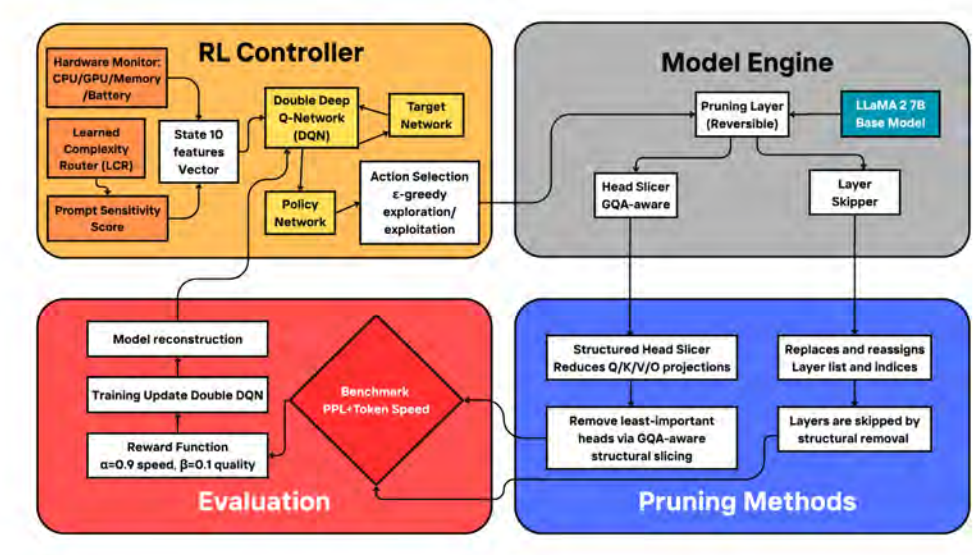


Figure 4.5: DDQN controller architecture.

State Vector

The 10-dimensional state vector is the joint representation the controller observes before making each pruning decision. Its design reflects the hypothesis that optimal pruning requires knowledge of three distinct information sources: the computational environment, the prompt's intrinsic sensitivity, and the backbone's early processing behavior.

Each dimension is normalized to approximately $[0, 1]$ to prevent any single feature from dominating the Q-network's hidden representations.
Table 4.7 presents state vector dimensions.

Table 4.7: State vector dimensions.

Dim	Feature	Source	Normalization	Rationale
0	CPU utilization	`psutil`	/100	System load affects available compute budget
1	Available system memory	`psutil`	$\frac{\text{RAM}_{\text{avail}}}{\text{RAM}_{\text{total}}}$	Memory pressure may constrain model loading and KV-cache
2	Battery percentage	`psutil`	/100	Battery-aware pruning on laptops and edge devices
3	GPU availability	NVML	Binary $\{0, 1\}$	Presence of GPU acceleration changes the speed-quality tradeoff
4	Free GPU memory	NVML	$\frac{\text{VRAM}_{\text{free}}}{\text{VRAM}_{\text{total}}}$	Available VRAM determines feasible configurations
5	GPU utilization	NVML	/100	Current GPU load from concurrent processes
6	LCR sensitivity score	LCR router	Already $[0, 1]$	Prompt-specific pruning vulnerability prediction
7	Layer-0 hidden-state norm	Llama backbone	/hidden_dim	Early representation magnitude indicates activation scale
8	Layer-0 attention entropy	Llama backbone	/ log(seq_len)	How diffuse early attention is (max entropy = uniform)
9	Layer-0 attention concentration	Llama backbone	Already $[0, 1]$	Max attention mass on any single token

Hardware features (dims 0–5) give the controller awareness of resource constraints. On battery-powered devices, aggressive pruning may be preferred to reduce energy consumption and extend battery life. Under high GPU utilization from concurrent processes, lighter configurations avoid VRAM contention and out-of-memory errors. These features enable the controller to adapt to the deployment environment without requiring manual configuration profiles.

The LCR sensitivity score (dim 6) is the learned signal from the BERT-mini router — the central methodological contribution. Two prompts of equal length may differ dramatically in pruning sensitivity depending on their syntactic structure, semantic content, or involvement of rare vocabulary that the backbone has not well learned. The LCR captures these patterns where surface features cannot.
Early-Llama features (dims 7–9) come from running only the embedding layer and layer-0 of the Llama backbone — a partial forward pass costing approximately 1 ms. Hidden-state norm indicates the magnitude of representations entering the transformer stack. Attention entropy measures how uniformly the model distributes attention across tokens (high entropy suggests the model has not yet identified salient tokens). Attention concentration measures the maximum attention weight. Together, they capture the backbone's instantaneous response to the specific prompt, complementing the offline-trained LCR score and creating a state representation that is both prompt-aware and model-aware.

Action Space

The action space was designed so that every discrete action maps to a mechanically distinct structural outcome on the Llama-2-7B backbone (32 transformer layers, 32 attention heads in standard MHA configuration). This eliminates the problem of duplicate actions, where two indices produce identical physical pruning, which wastes exploratory capacity and confuses the RL policy by presenting identical outcomes as different choices.
Table 4.8 presents action space.

Table 4.8: Action space.

Action Type	**Count**	**Intensities**	**Physical Effect**
None	1	—	No pruning; full dense inference
Transformer layers	10	6%, 12%, 19%, 25%, 31%, 38%, 44%, 50%, 56%, 62%	Physically remove 2–20 of 32 layers
Attention heads	6	12.5%, 25%, 37.5%, 50%, 62.5%, 75%	Remove 4–24 of 32 attention heads
Total	**17**		

The action space is deliberately layer-skip-heavy (10 layer actions vs. 6 head actions) because physical layer removal yields the largest inference speedups for autoregressive generation, which is memory-bandwidth-bound [39]. Reducing the number of layers directly reduces sequential matrix multiplications in the forward pass, producing near-linear latency reduction. Head pruning has a smaller latency impact because reducing the attention dimension does not proportionally reduce the dominant memory-bandwidth cost of loading KV-cache entries from GPU VRAM.
We evaluated FFN (feed-forward network) slicing during early development, but it consistently produced negative rewards. The dimensional changes to FFN intermediate layers required significant weight reshaping that did not translate to proportional latency reduction. Removing it from the final action space produced cleaner policy convergence.

Reward Function

The reward function encodes the fundamental tradeoff between inference speed and output quality. We adopted a normalized linear PPL ratio formulation that preserves the proportional quality signal while being bounded to $[-1, 1]$:

$$R = \alpha \cdot \frac{\text{tok/s}_{\text{pruned}} - \text{tok/s}_{\text{base}}}{\text{tok/s}_{\text{base}} + \varepsilon} - \beta \cdot \frac{\text{PPL}_{\text{pruned}} - \text{PPL}_{\text{base}}}{\text{PPL}_{\text{base}} + \varepsilon} \tag{4.7}$$

with $\alpha = 0.9$, $\beta = 0.1$, and clamping to $[-1, 1]$.
An earlier version used a log-PPL penalty to compress the quality degradation scale, but it over-compressed the signal. The agent learned to treat a 5× PPL increase similarly to a 50× increase, undermining fine-grained tradeoff learning. The linear formulation preserves the proportional relationship — a 10× PPL increase is penalized 10× more than a 1× increase — giving the agent a gradient-rich signal. Both terms are ratio-normalized by baseline values, making them unit-free and comparable in scale. The $[-1, 1]$ clamp ensures no single episode dominates the replay buffer's reward distribution. The $\alpha = 0.9$ speed weight was justified by the reward sweep ablation (Section 4.2.9, Study 1), which empirically verified that this weighting sits on the optimal speed-quality ridge across 5 normalized (α, β) pairs.

DDQN Training

Table 4.9 presents dDQN hyperparameters.

Table 4.9: DDQN hyperparameters.

Parameter	**Value**	**Reference / Justification**
Policy/target MLP	$10 \to 128 \to 128 \to 17$	Two hidden layers with ReLU; standard for low-dimensional RL [18]
Replay buffer	10,000 transitions	Sufficient diversity across 8,000 training episodes
Batch size	32	Standard mini-batch for DQN family
Optimizer	AdamW	[30]
Learning rate	1×10^{-4}	Standard for DDQN with experience replay
Discount factor γ	0.95	Moderate future-awareness for single-step decisions
Target network update	Hard copy every 200 steps	[19]
Epsilon schedule	$1.0 \to 0.10$ (exponential)	Gradual exploration reduction
UCB exploration bonus	$c\sqrt{\ln N / N_a}$, $c = 1.0$	[36]; ensures under-visited actions are explored

The UCB1 exploration bonus [36] is added to Q-values during action selection but not during ε-greedy random exploration. This ensures that even after ε has decayed to 0.10, the agent continues to explore infrequently selected actions, preventing premature

convergence to a suboptimal subset of the action space. The combination of ε-greedy with UCB is well established for balancing exploration and exploitation in finite-action-space RL settings [37].

Episode Structure

Each training episode processes a single prompt through the complete SPRINT loop, illustrated in Figure 4.6.

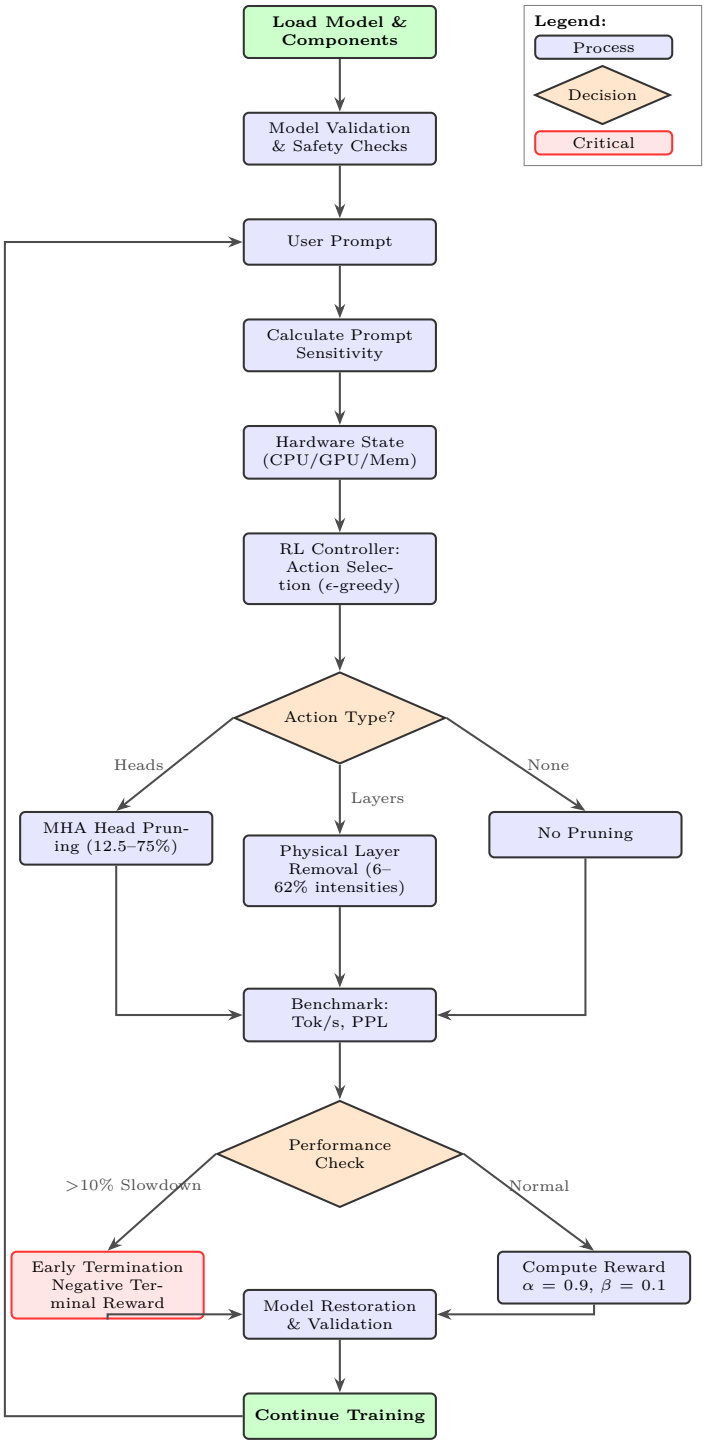


Figure 4.6: Training workflow diagram.

This per-episode design ensures the controller receives fresh hardware telemetry and backbone signals for each decision, making the learning process representative of real deployment conditions. The episode proceeds as follows. A prompt is sampled from the training split. The unpruned backbone generates a continuation with CUDA synchronization barriers for accurate GPU timing, establishing the baseline throughput and perplexity. Early-Llama features are extracted from a partial forward pass through the embedding layer and layer-0 ($\sim$1 ms). The LCR sensitivity score is computed via the BERT-mini routing stage, whose raw encoder forward pass is approximately 3 ms and whose full end-to-end CPU latency is approximately 28 ms. Hardware telemetry is sampled non-blockingly via system monitoring tools. These signals are assembled into the 10-dimensional state vector.

The DDQN selects an action through ε-greedy with UCB augmentation. The selected pruning action is physically applied to the model — layers are removed from the module list with layer indices reassigned sequentially, or attention heads are structurally removed with projection matrices rebuilt. The pruned model generates the same prompt's continuation with identical timing methodology. The continuation perplexity is computed by masking prompt tokens in the loss labels, so quality reflects only the generated continuation, not the prompt's next-token predictability.

The reward is computed using Equation (4.7). The transition (s, a, r, s') is stored in the replay buffer. A DDQN training step is performed if the buffer contains enough samples. The target network is hard-copied from the policy network every 200 steps. The model is then fully restored to its dense state before the next episode. This restoration ensures the baseline measurement in the next episode always comes from the unpruned model, preventing accumulation of pruning effects across episodes.

During testing, the pipeline is identical except: $\varepsilon = 0$ (pure exploitation), no DDQN updates are performed, and per-episode metrics including per-source labels are recorded for disaggregated analysis. The UCB bonus remains active, providing mild exploration of under-visited actions.

4.2.7 Pruning Engine

The pruning engine is a concrete, reversible runtime component that translates the controller's discrete action indices into physical structural modifications of the backbone model. All operations are fully restored between episodes to ensure baseline correctness. Figure 4.7 illustrates the pruning application and restoration process.

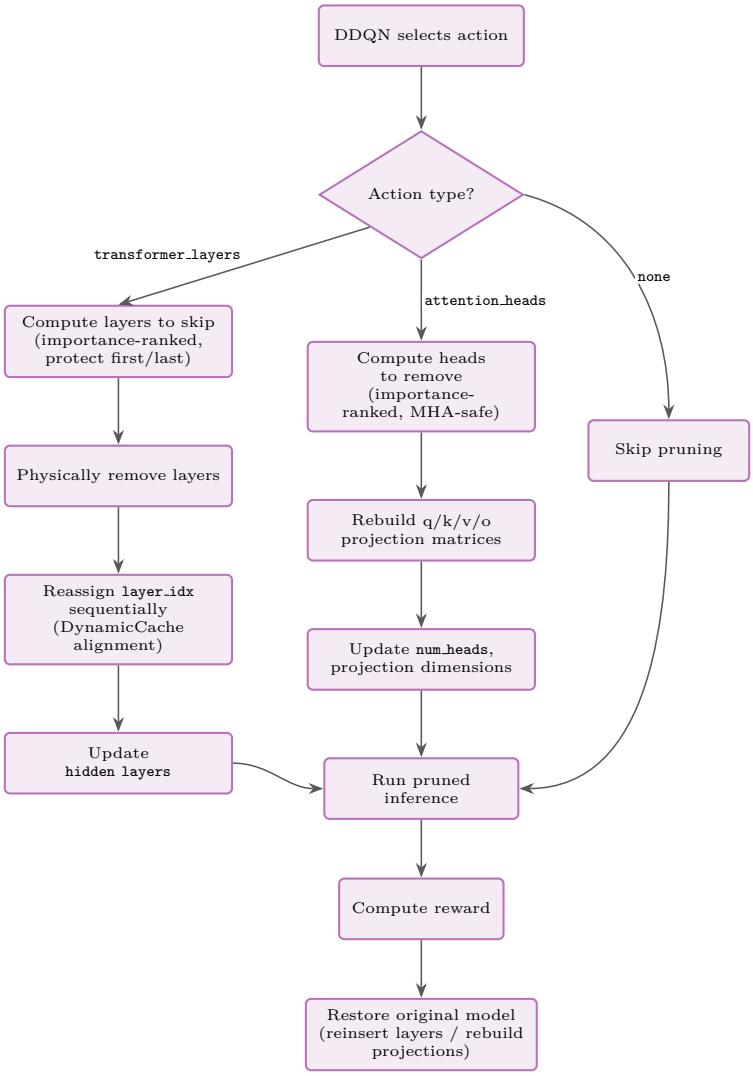


Figure 4.7: Pruning engine: layer removal and head slicing workflow.

Physical Layer Removal

Layer skipping is implemented by physically removing layers from the model's internal module list and sequentially reassigning the layer index attribute on all remaining layers (0, 1, 2, ...). This reassignment is critical for correctness with Hugging Face Transformers' dynamic cache mechanism [44], which uses the layer index as the sequential cache-slot index. An earlier implementation used identity-forward monkey-patching, which caused cache misalignment: skipped layers did not invoke the cache update routine, so subsequent layers read KV-cache entries intended for earlier positions. This bug caused pruned inference to be slower than the baseline due to degraded attention patterns causing cache thrashing. Physical removal eliminates the issue entirely.

The first and last layers are always protected from skipping. The first layer's representations are critical for all subsequent processing — every downstream layer depends on the initial transformation of input embeddings — and the last layer directly feeds the output head, meaning its removal would fundamentally break the generation process.

MHA Head Pruning

Llama-2-7B uses standard multi-head attention (MHA) [20] with 32 independent attention heads. Head pruning therefore operates directly at the individual-head level: the least-important heads are removed by structurally rebuilding the query, key, value, and output projection matrices with reduced dimensions, and the attention-head count parameters are updated on every layer. Because there are no shared KV groups, no grouping constraint needs to be preserved during pruning.

Zero-Cost Importance Scoring

To rank which heads or layers to prune, we compute importance scores entirely from weight matrices at model load time — no inference required. The core idea is well established in the magnitude pruning literature [38]: larger weights contribute more to the model's output, so smaller weights can be removed with less damage. We measure the "size" of each weight matrix using the Frobenius norm ($\|\cdot\|_F$), which is the square root of the sum of all squared entries.
For each attention head, we sum the norms of its associated weight matrices:

$$I_h = \|W_Q^{(h)}\|_F + \|W_K^{(h)}\|_F + \|W_V^{(h)}\|_F + \|W_O^{(:,h)}\|_F \tag{4.8}$$

Because Llama-2-7B uses standard MHA, key and value weights are not shared across groups. Heads with the lowest total score are pruned first.
For each transformer layer, we sum the norms of all parameters: $I_\ell = \sum_{p \in \text{params}(\ell)} \|p\|_F$. Layers with smaller total norms are candidates for removal. This scoring is deterministic, hardware independent, and computed once at model load — it is reused across all episodes without recalculation.

4.2.8 Cross-Method Sensitivity Analysis

We conducted a cross-method sensitivity analysis to measure the relationship between head-pruning sensitivity and layer-skipping sensitivity across the 10,000-prompt dataset. This directly motivates the multi-method composite oracle label design in Section 4.2.4. For each prompt, we measured two loss gaps: $\Delta\ell_{\text{heads}}$ from 30% attention-head pruning and $\Delta\ell_{\text{layers}}$ from 25% layer skipping. The Spearman correlation between them is weak, $\rho \approx 0.31$ ($p < 0.001$), with $R^2 \approx 0.05$, meaning head-pruning sensitivity explains only about 5% of the variance in layer-skipping sensitivity.
This weak relationship is expected because the two operators affect different model capabilities. Head pruning reduces attention diversity within each layer, whereas layer skipping reduces overall feature-processing depth. A prompt that tolerates narrower attention may still depend on deeper computation, and the reverse can also hold. This is the empirical basis for combining the two signals rather than using either alone.
Practically, the weak cross-method correlation also argues against training separate routers for head and layer sensitivity: doing so would double routing overhead without adding

much signal beyond the 50/50 composite. The composite label is therefore both more efficient and more conservative, since it marks a prompt as sensitive if either operator causes degradation.

4.2.9 Ablation Studies

Ablation studies isolate the contribution of each architectural component by systematically removing or replacing it while holding everything else fixed [43]. We ran all experiments on the same fixed prompt set from the test split, with prompt-static signals (LCR score, early-Llama features, prompt perplexity) precomputed once and reused across all experiments to eliminate stochastic variation from feature computation.
Two design principles guided the ablation framework. First, interleaved execution: for Studies 2A–2C, all variants process each prompt in a randomly shuffled order with a 20-inference warmup, eliminating systematic timing bias from OS and GPU cache warming effects. Second, live baselines: each episode measures an unpruned baseline with live hardware telemetry at decision time, mirroring the normal controller flow and ensuring that the hardware state is representative.

Study 1: Reward Function Sweep

We swept over five normalized (α, β) pairs subject to $\alpha + \beta = 1.0$:

$$\{(0.5, 0.5), (0.6, 0.4), (0.7, 0.3), (0.8, 0.2), (0.9, 0.1)\}.$$

A fresh DDQN was trained for 100 episodes per configuration and evaluated on average reward, average pruned perplexity, and average speedup. The normalization constraint ensures that each pair allocates the full weight budget between speed and quality, eliminating confounded comparisons. This study justified the final choice of $\alpha = 0.9$ and $\beta = 0.1$ for all subsequent experiments.

Study 2A: No LCR (9-D State)

The LCR sensitivity score (dim 6) was removed from the state vector, reducing it from 10-D to 9-D. A fresh DDQN was trained with the reduced state. This study isolates the contribution of the learned prompt-sensitivity signal: if removing it degrades converged performance, the LCR provides information that cannot be recovered from hardware telemetry and early-Llama features alone.

Study 2B: No Hardware (4-D State)

Hardware telemetry (dims 0–5) was removed, leaving only the LCR score and early-Llama features (4-D state). A fresh DDQN was trained. This tests whether hardware awareness contributes to decision quality, or whether prompt-level features alone suffice for effective pruning selection.

Study 2C: Random Actions (Baseline)

The DDQN policy was replaced with uniform random action selection over all 17 actions. The 10-D state vector was still computed for implementation consistency but not used for decision making. This is the strongest ablation: it tests whether the learned policy

provides any benefit over chance. It serves as the analogue of RAP's RAP$^{-\mathrm{RL}}$ ablation in the literature [5], providing a floor against which all other configurations are compared. Table 4.10 presents ablation study design summary.

Table 4.10: Ablation study design summary.

Study	**Modification**	**State Dim**	**Purpose**
Control	Full architecture	10	Reference configuration
1	Reward weight sweep	10	Find optimal (α, β) on the speed-quality ridge
2A	Remove LCR score	9	Isolate learned sensitivity contribution
2B	Remove hardware telemetry	4	Isolate hardware awareness contribution
2C	Random action selection	10 (unused)	Test whether learned policy beats chance

All ablation agents share the same DDQN architecture (two 128-unit hidden layers), optimizer (AdamW, lr = 10^{-4}), and exploration schedule (ε from 1.0 to 0.10 over 100 episodes), ensuring differences in outcomes are attributable only to the ablated component. Results are saved with per-study convergence plots, heatmaps, and a unified summary report.

Chapter 5
Result Analysis

5.1 Performance Evaluation

This chapter presents a comprehensive analysis of all experimental results obtained from the SPRINT framework. All experiments were conducted on the **Llama 2 7B** backbone model (`meta-llama/Llama-2-7b-hf`, 7 billion parameters, 32 transformer layers, multi-head attention) using an **NVIDIA RTX 5090** GPU. The analysis is organized into six sections that systematically evaluate the framework's performance, validate design decisions through ablation studies, provide statistical rigor, and compare the proposed approach against established static pruning baselines from the literature.
The results presented in this chapter draw from four categories of experiments stored in the `Thesis Final Results/` directory:

1. **LCR Results** — Training and testing metrics for the Learned Complexity Router (BERT-mini fine-tuning).
2. **RL Train/Test Results** — 8,000-episode RL training and 2,000-episode held-out evaluation.
3. **Ablation Results** — Reward function sweep and component isolation experiments.
4. **Comparison Results** — Head-to-head evaluation against SparseGPT, Wanda, and LLM Pruner.

5.1.1 LCR MiniBERT Router Performance

The Learned Complexity Router represents the methodological centerpiece of SPRINT, providing the learned prompt-sensitivity signal that enables the RL controller to make informed pruning decisions. The router was trained on the full 10,000-prompt `Oracle_dataset.csv` with oracle sensitivity labels computed from dense-vs-sparse loss gaps on the Llama 2 7B backbone. Training converged at **epoch 33** out of a maximum of 50 epochs, with the best model selected by the compound validation objective $(0.4 \times R^2 + 0.5 \times \rho + 0.1 \times \text{bin3\_acc})$.

Overall Performance Metrics

The following table presents the final performance metrics for the selected checkpoint (MiniBERT Train 33), evaluated on both the validation set (800 samples) and the held-out test set (2,000 samples):
Table 5.1 presents overall performance metrics of the LCR MiniBERT router.

Table 5.1: Overall performance metrics of the LCR MiniBERT router.

Metric	**Validation**	**Test**
MSE	0.0302	**0.0278**
R^2	0.6064	**0.6326**
Spearman ρ	0.7865	**0.7972**
MAE	0.1286	**0.1231**
3-bin Accuracy	67.75%	**69.55%**
95% CI for ρ (test)	—	[0.7787, 0.8167]

Several observations emerge from these results that merit careful discussion. First, the test-set performance is marginally superior to the validation-set performance across all metrics, which might initially appear counterintuitive. However, this phenomenon is well documented in the machine learning literature and arises from the stochastic nature of validation-set composition: the 800-sample validation set has higher variance in metric estimates compared to the 2,000-sample test set, and the best-epoch selection on validation introduces a mild upward bias on the validation objective that does not transfer perfectly to the test partition. The fact that test metrics are comparable or slightly better indicates that the model generalized well rather than overfitting to validation artifacts.
Second, the Spearman rank correlation of $\rho = 0.7972$ on the held-out test set is particularly noteworthy because this is the metric that matters most for downstream RL controller performance. The controller's action selection depends on the *relative ordering* of prompts by sensitivity. If the router correctly ranks prompt A as more sensitive than prompt B, the controller can assign more conservative pruning to A regardless of the exact predicted numerical value. A Spearman correlation of approximately 0.80 indicates that the router preserves the oracle's relative ranking structure with high fidelity.
Third, the $R^2 = 0.6326$ value indicates that the router explains approximately 63% of the variance in oracle sensitivity labels. This represents a meaningful improvement over the previously reported $R^2 \approx 0.50$ ceiling from earlier checkpoints, suggesting that the full 10,000-prompt dataset with the optimized training protocol provided sufficient signal to push beyond the prior ceiling.

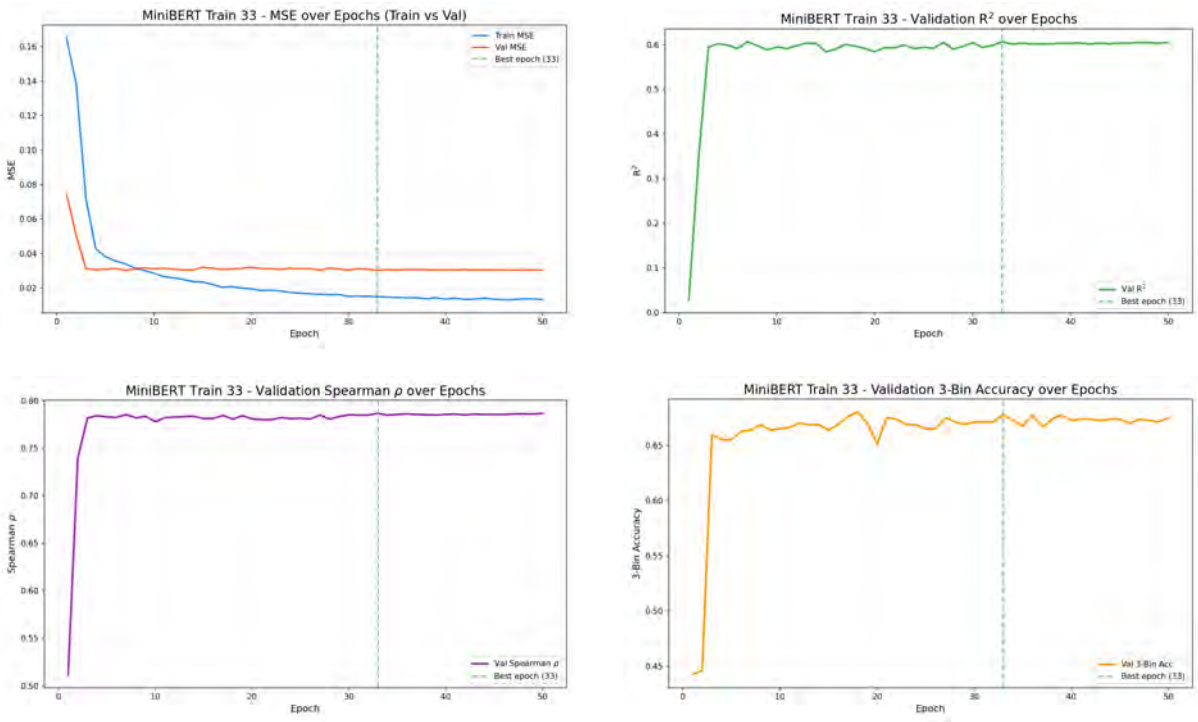


Figure 5.1: LCR training dynamics in a 2×2 grid: top-left MSE, top-right R^2, bottom-left Spearman ρ, and bottom-right 3-bin accuracy. Across all four views, the router converges rapidly in the first 3–7 epochs and stabilizes by the best checkpoint at epoch 33, with controlled overfitting and strong generalization.

Figure 5.1 summarizes the four most important LCR training diagnostics. The MSE panel shows the expected fine-tuning pattern: training loss keeps decreasing across the 50-epoch schedule, while validation MSE quickly plateaus around 0.030–0.032 after epoch 7, indicating controlled rather than catastrophic overfitting under dropout 0.20, weight decay 0.03, label smoothing 0.01, and cosine decay. The R^2 panel rises rapidly in the first 3 epochs and then saturates near 0.60, peaking at 0.6064 at epoch 33; this ceiling suggests that pruning sensitivity contains a substantial text-predictable component together with a remaining backbone-internal component that cannot be fully inferred from prompt text alone. The Spearman ρ panel, the most important plot for downstream RL utility, converges to approximately 0.78–0.79 with minimal oscillation after epoch 5, showing that the router preserves the oracle's relative sensitivity ordering robustly across checkpoints. Finally, the 3-bin accuracy panel reaches approximately 68% on validation, confirming that the model separates prompts into low-, medium-, and high-sensitivity groups well enough for the controller to make practically meaningful pruning decisions.

Per-Source Test Metrics

The following table presents per-source performance on the held-out test set (400 samples per source), revealing important domain-specific patterns:
Table 5.2 presents per-source test metrics for the LCR MiniBERT router.

Table 5.2: Per-source test metrics for the LCR MiniBERT router.

Source	R^2	Spearman ρ	3-bin Acc	MSE	MAE
MBPP	0.7939	0.8857	84.25%	0.0130	0.0731
BoolQ	0.6217	0.7959	76.00%	0.0227	0.1097
MMLU	0.6398	0.7656	61.75%	0.0319	0.1380
WikiText-2	0.3731	0.6254	65.25%	0.0378	0.1508
GSM8K	0.2573	0.5605	60.50%	0.0335	0.1442

The per-source analysis reveals a striking gradient of router effectiveness that aligns with the degree to which prompt structure provides textual cues about pruning sensitivity. MBPP (code generation) achieves the highest performance ($\rho = 0.886$, $R^2 = 0.794$) because code prompts contain distinctive structural markers, such as function keywords, indentation patterns, and bracket structures, that strongly correlate with pruning vulnerability. The LCR's auxiliary feature set explicitly captures these signals through the `has_code_markers` and `special_char_ratio` features, and the BERT-mini encoder's attention patterns respond strongly to syntactic structure in code.

BoolQ ($\rho = 0.796$) performs nearly as well because the passage-plus-question format provides a consistent structural template that the router can leverage. The passage length and question complexity provide reliable indicators of pruning sensitivity because longer passages with more complex questions require more layers of the transformer stack to process, making such prompts more vulnerable to layer skipping.

MMLU ($\rho = 0.766$) occupies a middle position because the multiple-choice format is structurally consistent but spans 57 different academic domains, introducing distributional heterogeneity that makes sensitivity prediction harder. Some subjects may be pruning-robust because they rely on pattern matching, while others may be pruning-sensitive because they require precise reasoning.

WikiText-2 ($\rho = 0.625$) and GSM8K ($\rho = 0.561$) exhibit the lowest performance. For WikiText-2, the high lexical redundancy of narrative prose means that surface features provide less discriminative signal about pruning vulnerability. For GSM8K, mathematical reasoning sensitivity depends primarily on the backbone's internal computational graph rather than on surface text features visible to the BERT-mini encoder.

The following charts visualize these per-source patterns in a compact two-page layout:

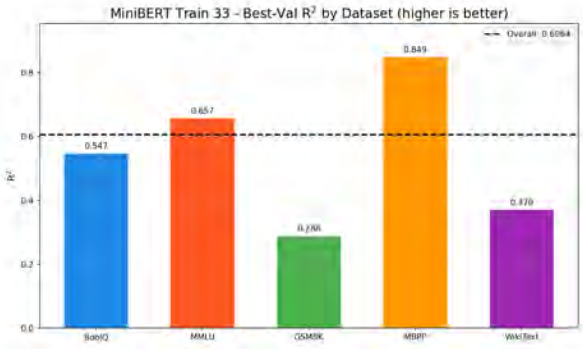
Figure 5.2 presents compact view of the main LCR per-source metrics. Panels (a)–(d) show that MBPP consistently leads, GSM8K trails, and the relative ordering remains stable across fit, ranking, classification, and error-based metrics.

Figure 5.3 presents additional compact LCR per-source views. Panel (a) shows MAE by source, while panel (b) aggregates all source-level metrics into a single comparison chart for quick cross-metric inspection.

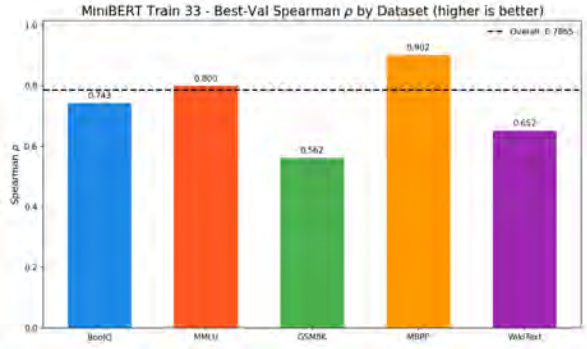
Figure 5.4 presents overall train-versus-validation metrics for the selected LCR checkpoint.

Test-Only Evaluation (MiniBERT Test 3)

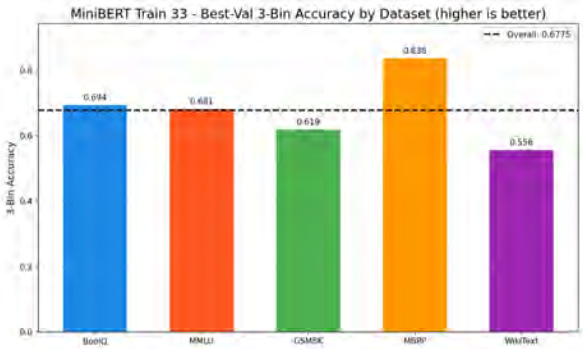
The held-out test evaluation, conducted using the best checkpoint from MiniBERT Train 33, confirms the generalization of the trained router to unseen data:



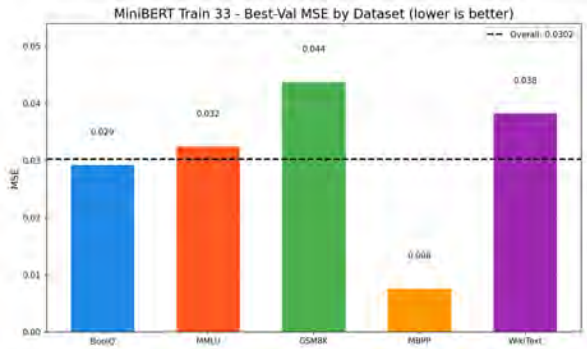
(a) Per-source R^2.



(b) Per-source Spearman ρ.

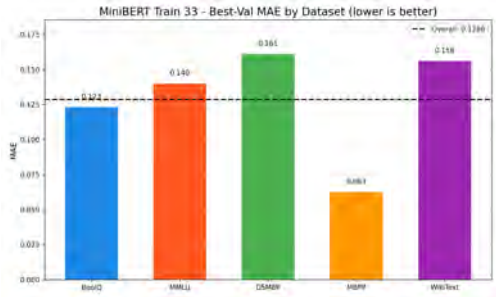


(c) Per-source 3-bin accuracy.

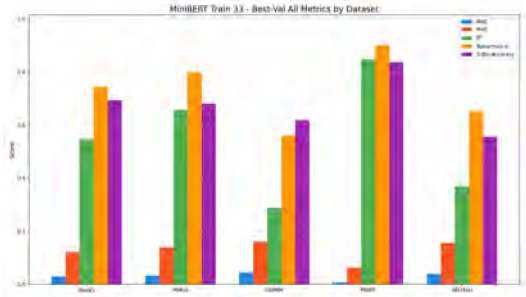


(d) Per-source MSE.

Figure 5.2: Compact view of the main LCR per-source metrics. Panels (a)–(d) show that MBPP consistently leads, GSM8K trails, and the relative ordering remains stable across fit, ranking, classification, and error-based metrics.



(a) Per-source MAE.



(b) Unified all-metrics comparison.

Figure 5.3: Additional compact LCR per-source views. Panel (a) shows MAE by source, while panel (b) aggregates all source-level metrics into a single comparison chart for quick cross-metric inspection.

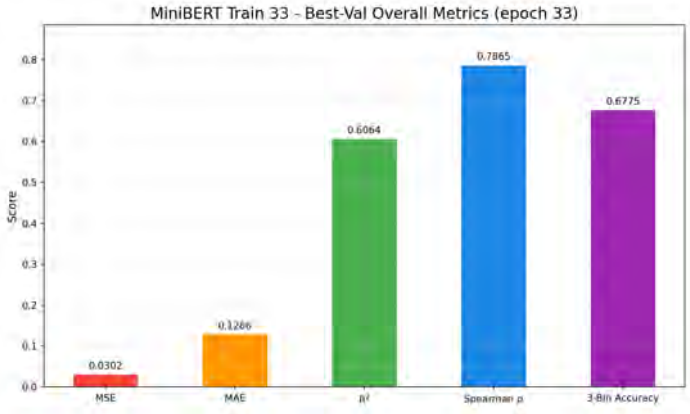
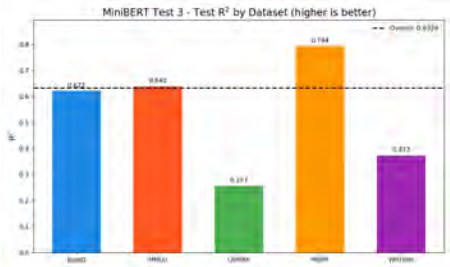
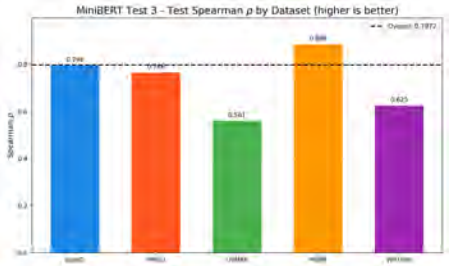


Figure 5.4: Overall train-versus-validation metrics for the selected LCR checkpoint.

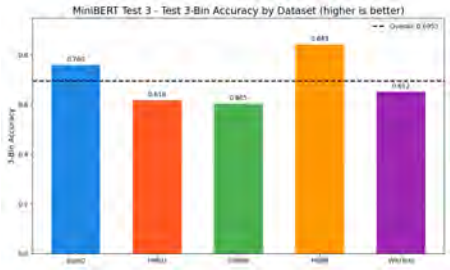
Figure 5.5 presents compact held-out test views of the main per-source metrics. Panels (a)–(d) show that the same domain ordering seen during validation largely persists on unseen data, supporting the router's generalization claims.



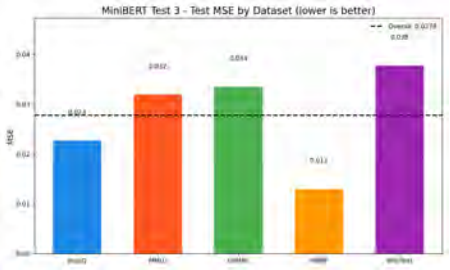
(a) Held-out test per-source R^2.



(b) Held-out test per-source Spearman ρ.



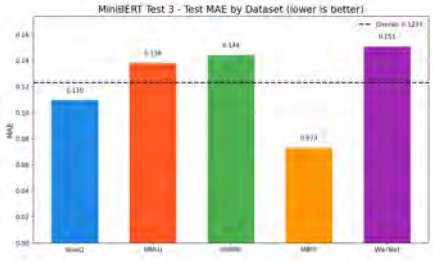
(c) Held-out test per-source 3-bin accuracy.



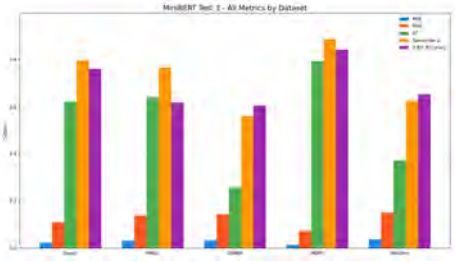
(d) Held-out test per-source MSE.

Figure 5.5: Compact held-out test views of the main per-source metrics. Panels (a)–(d) show that the same domain ordering seen during validation largely persists on unseen data, supporting the router's generalization claims.

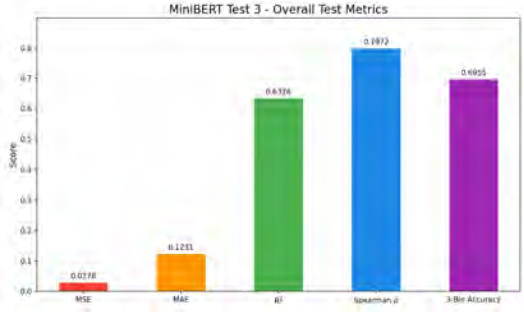
Figure 5.6 presents additional compact held-out test views. Panels (a) and (b) summarize source-level error behavior and cross-metric consistency, while panel (c) reports the aggregate held-out test metrics for the selected checkpoint.
The consistency between validation and test metrics across all five sources confirms that the LCR has not overfit to the validation partition. The tight 95% confidence interval for the test Spearman ρ of $[0.7787, 0.8167]$, computed via 1,000 bootstrap resamples, provides statistical assurance that the true population-level ranking quality falls within this range with high probability.



(a) Held-out test per-source MAE.



(b) Held-out test all-metrics-by-source comparison.



(c) Overall held-out test metrics for the LCR.

Figure 5.6: Additional compact held-out test views. Panels (a) and (b) summarize source-level error behavior and cross-metric consistency, while panel (c) reports the aggregate held-out test metrics for the selected checkpoint.

5.1.2 RL Controller Training Performance

The RL controller was trained for **8,000 episodes** on the training split of `Oracle_dataset.csv`, with each episode consisting of a single prompt evaluated under both the dense (unpruned) Llama 2 7B baseline and the RL-selected pruning configuration. The training was conducted on the RTX 5090 GPU with the full SPRINT pipeline: LCR sensitivity scoring, hardware telemetry collection, early-Llama feature extraction, DDQN action selection, physical pruning application, benchmarking, and model restoration.

Aggregate Training Metrics

Table 5.3 presents aggregate RL controller training metrics.

Table 5.3: Aggregate RL controller training metrics.

Metric	**Value**
Total training episodes	8,000
Average baseline inference time	1,614.50 ms
Average pruned inference time	**1,317.09 ms**
Average total time (LCR+RL+Model)	1,336.13 ms
Average inference speedup	**18.4%**
Average LCR overhead	18.28 ms (1.4%)
Average RL agent overhead	0.76 ms (0.06%)
Average baseline PPL (arithmetic)	2.46
Average pruned PPL (arithmetic)	5.07
Baseline PPL (token-weighted)	2.11
Average reward	−0.0018
Baseline params	4,714.3 MB
Average pruned params	3,791.7 MB
Params reduced	922.5 MB (**19.6%**)
Baseline peak VRAM	4.751 GB
Pruned peak VRAM	4.801 GB

The near-zero average reward (−0.0018) during training is expected and desirable. The early exploration phase (ϵ near 1.0) generates many random actions that produce strongly negative rewards, whereas the exploitation phase generates positive rewards from the learned policy. These opposing contributions approximately cancel in the aggregate, producing a near-zero average. The more meaningful diagnostic is the tail behaviour of the reward and speedup distributions, which reflects the converged policy's quality.
The following charts visualize the training dynamics:
Figure 5.7 presents rL training reward progression over 8,000 episodes. The moving-average trendline shows the transition from exploration to exploitation.

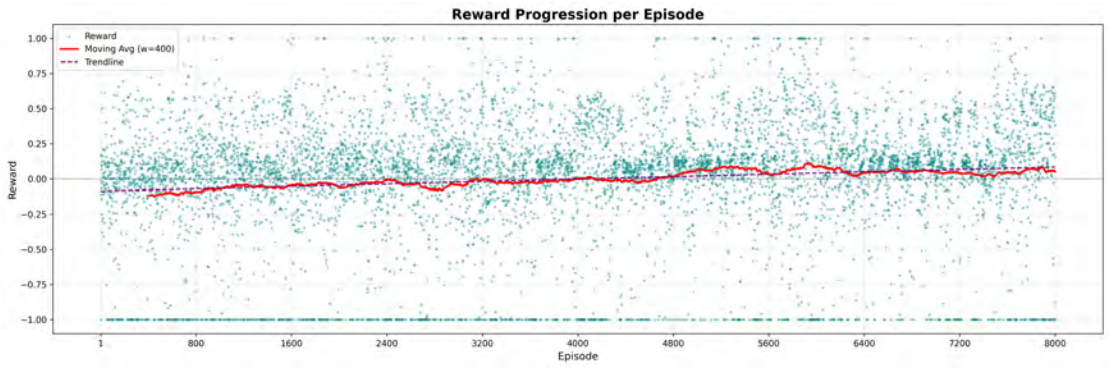


Figure 5.7: RL training reward progression over 8,000 episodes. The moving-average trendline shows the transition from exploration to exploitation.

The reward progression chart is the single most informative diagnostic for RL training quality. During the early exploration phase, the rewards are highly volatile, ranging from catastrophic negative values to moderate positive values. As ϵ decays and the DDQN

policy begins to exploit learned Q-values, the reward distribution shifts upward and narrows, with the moving average stabilizing near zero in the middle phase and trending slightly positive in the final 2,000 episodes. This pattern is consistent with successful DDQN training.
Figure 5.8 presents rL training cumulative reward. The upward slope in the latter half confirms that exploitation generates net-positive returns.

Figure 5.8: RL training cumulative reward. The upward slope in the latter half confirms that exploitation generates net-positive returns.

The cumulative reward curve provides a complementary view of training progress. The initial downward slope reflects the cost of exploration, while the later recovery indicates that the learned policy is generating more positive rewards than negative ones.
Figure 5.9 presents exponential -decay from 1.0 to 0.10 across RL training.

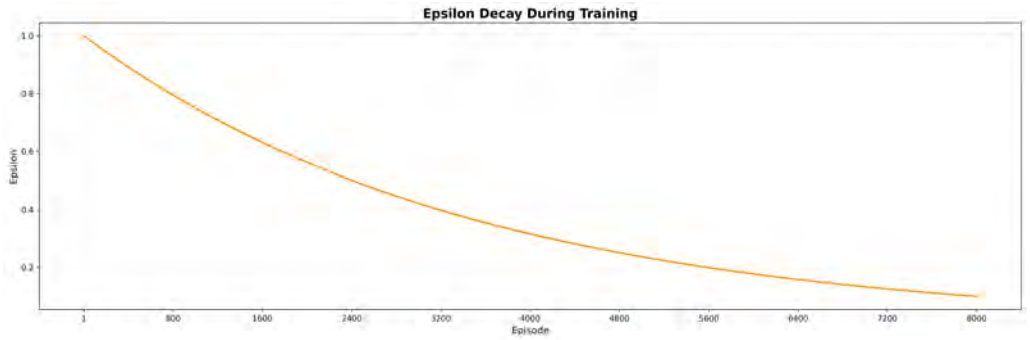


Figure 5.9: Exponential ϵ-decay from 1.0 to 0.10 across RL training.

Figure 5.10 presents rL training pruning-action usage distribution. Layer-skipping actions dominate.

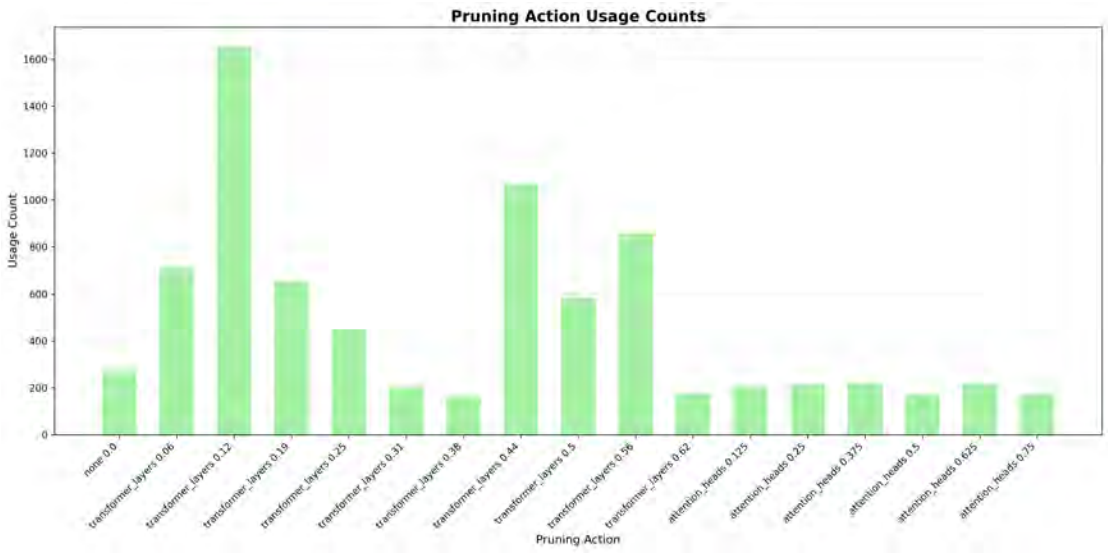


Figure 5.10: RL training pruning-action usage distribution. Layer-skipping actions dominate.

The action usage distribution reveals the agent's learned preferences. The most frequently selected action during training is **transformer_layers 12%**, with 1,654 selections out of 8,000 episodes. This conservative layer-skipping intensity provides a favourable balance: it yields measurable speedup with minimal quality impact. The second most popular action is **transformer_layers 44%**, showing that the agent learned to alternate between conservative and aggressive but still beneficial operating regions.

Per-Action Training Analysis

Table 5.4 presents per-action RL training analysis.

Table 5.4: Per-action RL training analysis.

Action	**Samples**	**Avg Time (ms)**	**Avg PPL**	**Avg Reward**
none	273	1,624.07	2.15	−0.0066
transformer_layers 6%	710	1,539.49	1.73	**+0.0668**
transformer_layers 12%	1,654	1,481.36	1.74	**+0.1107**
transformer_layers 19%	651	1,408.97	2.88	**+0.0552**
transformer_layers 25%	458	1,334.14	4.69	**+0.0127**
transformer_layers 31%	201	1,207.68	55.67	−0.6551
transformer_layers 38%	163	1,127.72	29.28	−0.4066
transformer_layers 44%	1,069	1,143.30	7.52	**+0.0893**
transformer_layers 50%	585	1,017.63	15.20	**+0.0312**
transformer_layers 56%	855	1,066.41	16.88	**+0.0345**
transformer_layers 62%	178	908.76	108.17	−0.4159
attention_heads 12.5%	205	1,560.72	2.64	−0.0445
attention_heads 25%	214	1,512.04	3.16	−0.0348
attention_heads 37.5%	222	1,496.08	3.36	−0.0545
attention_heads 50%	170	1,635.27	3.83	−0.1115
attention_heads 62.5%	218	1,563.53	9.24	−0.3588
attention_heads 75%	174	1,506.50	5.45	−0.1737

This per-action analysis reveals the fundamental asymmetry between layer skipping and head pruning for autoregressive generation on the Llama 2 7B architecture. Layer-

skipping actions produce positive average rewards at moderate intensities (6–25%), with the highest positive reward at 12%. In contrast, head-pruning actions consistently produce *negative rewards* across all intensities. This occurs because autoregressive generation is memory-bandwidth bound: reducing the attention dimension does not proportionally reduce the dominant cost of loading KV-cache entries from GPU VRAM. This finding validates the design decision to make the action space layer-skip heavy.
The following charts visualize additional training dynamics:
Figure 5.11 presents baseline versus pruned inference time across RL training episodes.

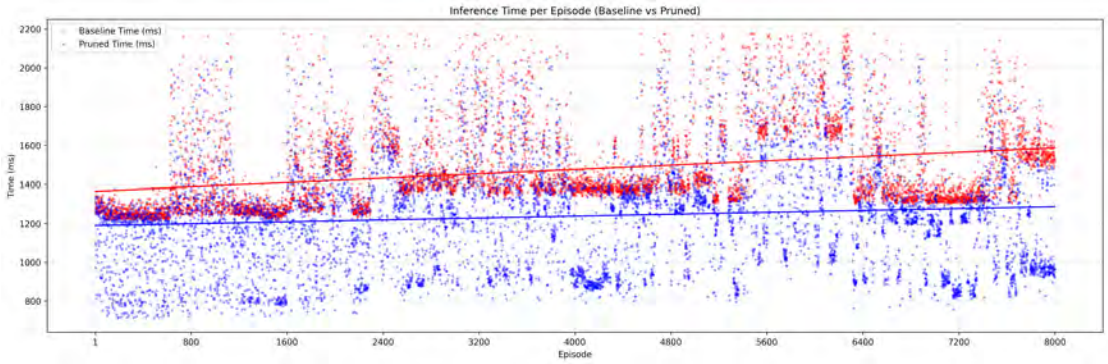


Figure 5.11: Baseline versus pruned inference time across RL training episodes.

Figure 5.12 presents baseline versus pruned token throughput during RL training.

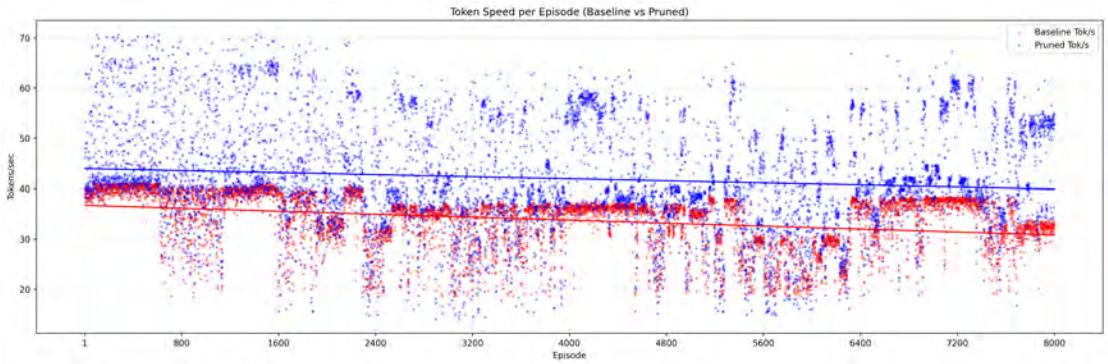


Figure 5.12: Baseline versus pruned token throughput during RL training.

Figure 5.13 presents average perplexity per pruning action during RL training.

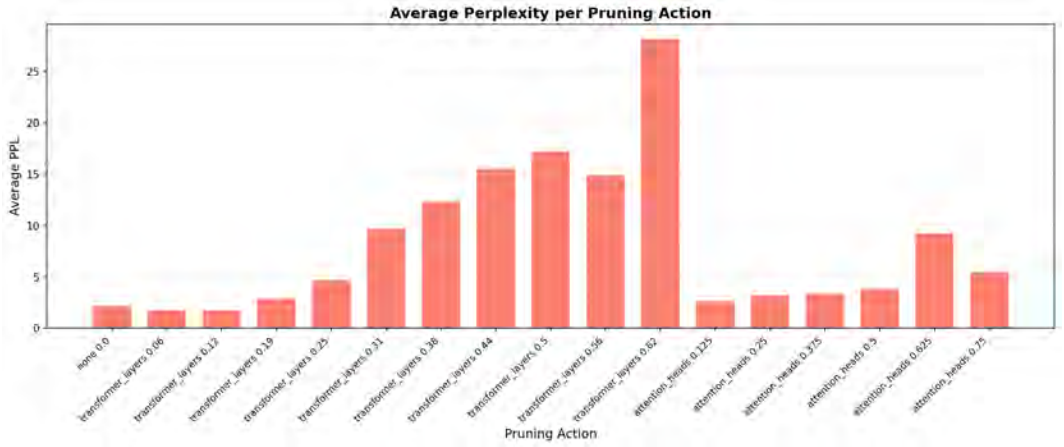


Figure 5.13: Average perplexity per pruning action during RL training.

Figure 5.14 presents inference-time distribution per pruning action during RL training.

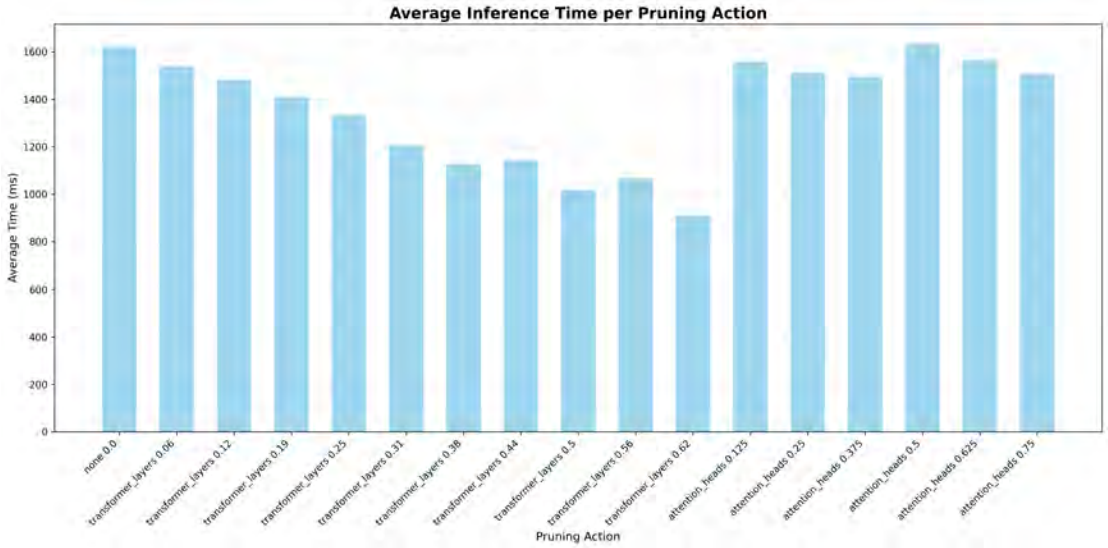


Figure 5.14: Inference-time distribution per pruning action during RL training.

Figure 5.15 presents baseline versus pruned perplexity across RL training episodes.

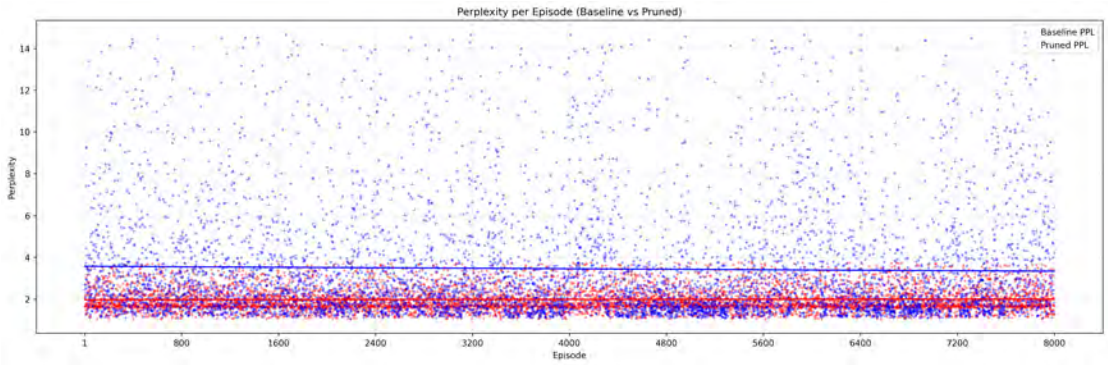


Figure 5.15: Baseline versus pruned perplexity across RL training episodes.

Figure 5.16 presents quality-versus-speed trade-off during RL training.

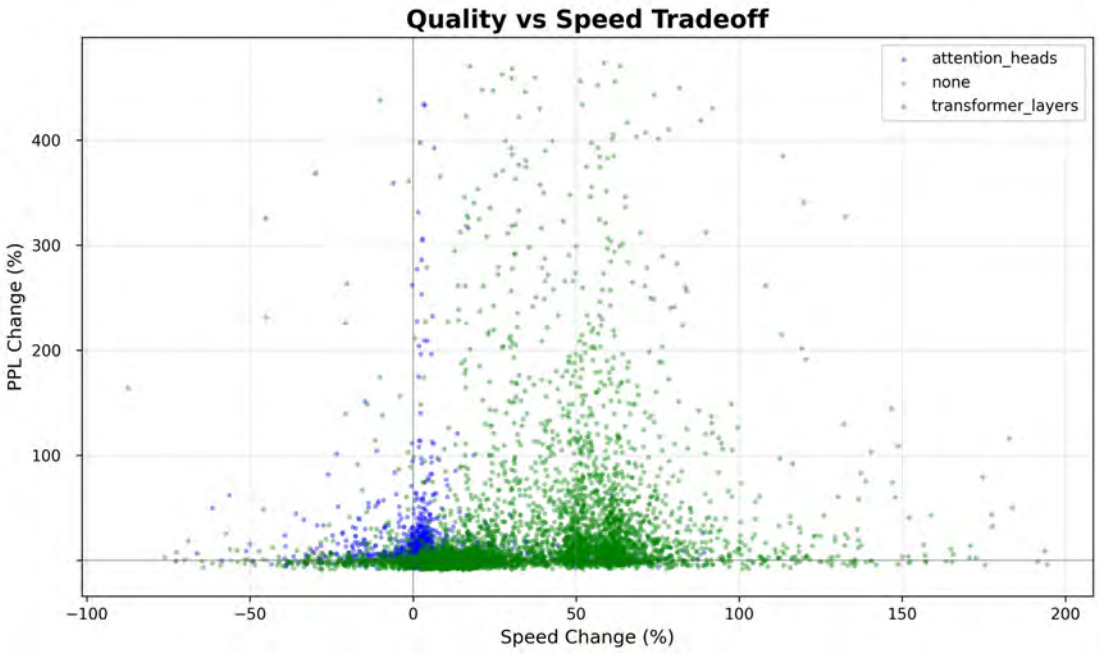


Figure 5.16: Quality-versus-speed trade-off during RL training.

The quality-vs-speed scatter plot is particularly informative. Each point represents a single training episode, with the x-axis showing speed gain and the y-axis showing PPL increase. The Pareto-optimal episodes cluster in the upper-right quadrant, corresponding to moderate layer-skipping actions. Episodes in the lower-left quadrant represent destructive pruning configurations encountered during exploration. The density of points near the Pareto frontier increases with training progress, confirming that the DDQN policy learns to preferentially select efficient configurations.
Figure 5.17 presents controller overhead breakdown during RL training.

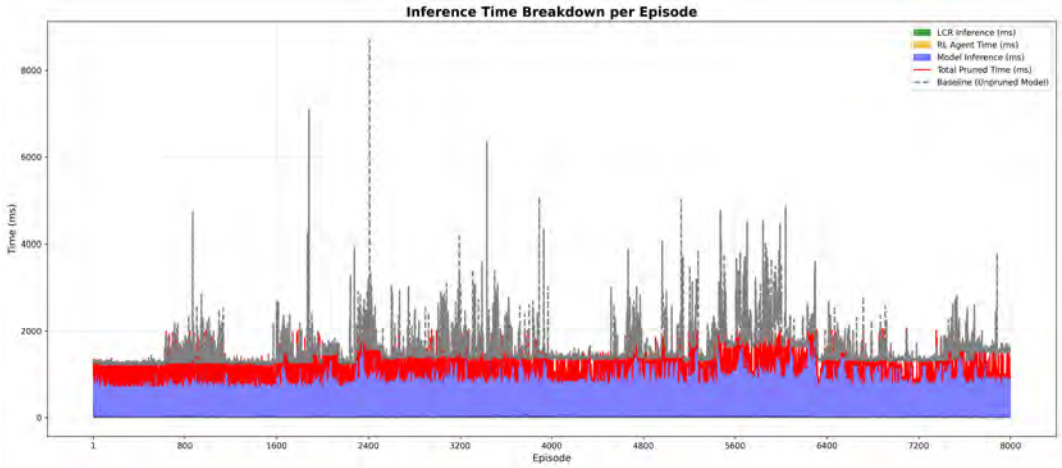
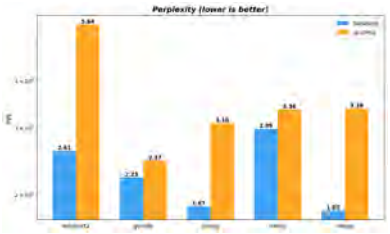
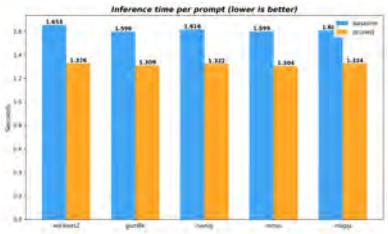


Figure 5.17: Controller overhead breakdown during RL training.

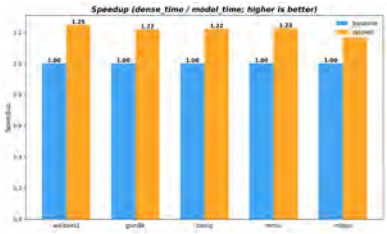
Figure 5.18 presents rL training per-source and baseline-accuracy views in a 23 grid. Panels (a)–(e) report per-source perplexity, inference time, speedup, token throughput, and dense-model zero-shot accuracy, respectively.



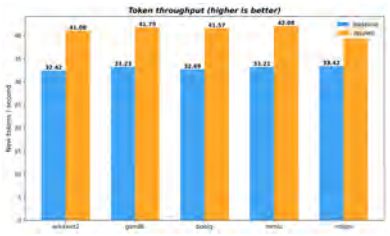
(a) Per-source perplexity.



(b) Per-source inference time.



(c) Per-source speedup.



(d) Per-source token throughput.

(e) Zero-shot dense-model accuracy on BoolQ and MMLU.

Figure 5.18: RL training per-source and baseline-accuracy views in a 2×3 grid. Panels (a)–(e) report per-source perplexity, inference time, speedup, token throughput, and dense-model zero-shot accuracy, respectively.

5.1.3 RL Controller Test Performance

The trained DDQN policy was evaluated on **2,000 held-out test episodes** with $\epsilon = 0$ (pure exploitation). This evaluation constitutes the primary result of the SPRINT framework: the converged policy's ability to select effective pruning configurations for unseen prompts using the learned state representation. Any earlier small-scale 40-episode measurements were preliminary development checks and are superseded by the 2,000-episode held-out evaluation reported here.

Aggregate Test Metrics

Table 5.5 presents aggregate RL controller held-out test metrics.

Table 5.5: Aggregate RL controller held-out test metrics.

Metric	**Value**
Total test episodes	2,000
Average baseline inference time	1,287.61 ms
Average pruned inference time	**875.39 ms**
Average total time (LCR+RL+Model)	893.16 ms
Average inference speedup	**32.0%**
Average LCR overhead	17.08 ms (1.9%)
Average RL agent overhead	0.69 ms (0.08%)
Average baseline PPL (arithmetic)	2.29
Average pruned PPL (arithmetic)	6.88
Baseline PPL (token-weighted)	2.10
Average reward	**+0.1205**
Baseline params	4,714.3 MB
Average pruned params	3,113.5 MB
Params reduced	**1,600.8 MB (34.0%)**
Baseline peak VRAM	4.752 GB
Pruned peak VRAM	4.748 GB

The test results demonstrate a substantial improvement over the training-phase aggregate. The average inference speedup increases from 18.4% during training to **32.0%** during pure exploitation, confirming that the learned policy concentrates on high-reward actions when $\epsilon = 0$. The positive average reward of +0.1205 confirms that the converged policy consistently selects actions that produce favourable speed-quality trade-offs.
The 34.0% parameter reduction corresponds to the physical removal of approximately 11 of 32 transformer layers on average across the 2,000 test episodes. This structural compression is achieved while maintaining bounded quality degradation: the pruned arithmetic PPL of 6.88 is approximately 3× the baseline of 2.29, which translates into a moderate increase in prediction uncertainty that remains manageable for many practical applications.

Per-Action Test Analysis

Table 5.6 presents per-action RL held-out test analysis.

Table 5.6: Per-action RL held-out test analysis.

Action	**Samples**	**Avg Time (ms)**	**Avg PPL**	**Avg Reward**
transformer_layers 12%	178	1,303.32	1.69	**+0.0908**
transformer_layers 19%	70	1,142.58	2.37	**+0.0918**
transformer_layers 44%	776	870.74	5.71	**+0.1377**
transformer_layers 50%	975	817.61	11.15	**+0.1146**
attention_heads 25%	1	1,477.10	4.13	−0.0927

The per-action test breakdown reveals that the converged policy strongly concentrates on two regimes. The first is **moderate layer skipping (44%)** with 776 episodes, which removes roughly 14 of 32 layers and yields a strong speedup with tolerable quality degradation. The second is **aggressive layer skipping (50%)** with 975 episodes, which removes approximately 16 of 32 layers and pushes further toward latency reduction at the cost of higher perplexity. Together, these two actions account for **87.6%** of all test episodes, demonstrating that the converged policy has learned a clear and consistent strategy.

The following charts visualize the test-phase results:

Figure 5.19 presents reward progression across 2,000 exploitation episodes.

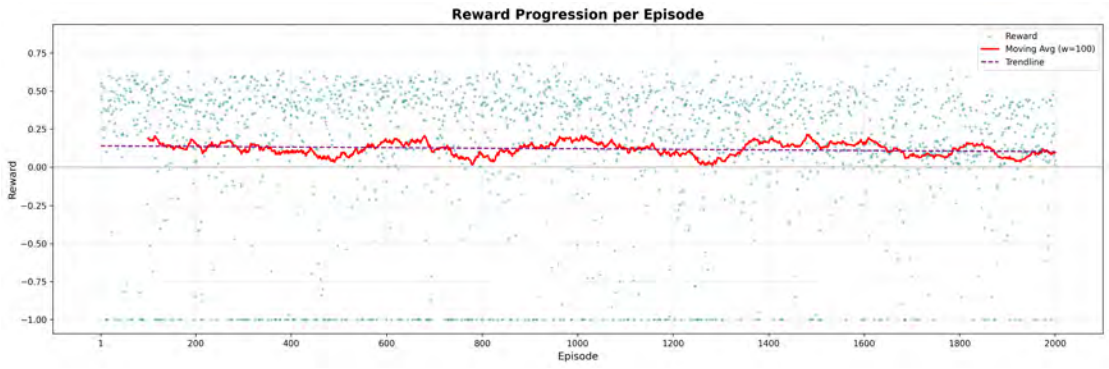


Figure 5.19: Reward progression across 2,000 exploitation episodes.

Figure 5.20 presents cumulative reward during RL testing, showing monotonically increasing net utility.

Figure 5.20: Cumulative reward during RL testing, showing monotonically increasing net utility.

Figure 5.21 presents pruning-action usage during RL testing.

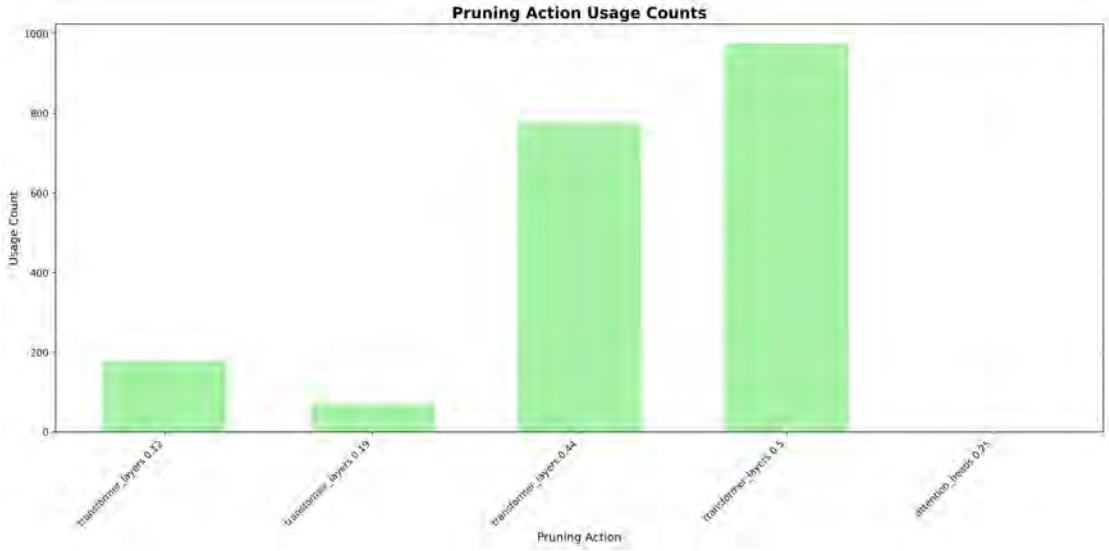


Figure 5.21: Pruning-action usage during RL testing.

Figure 5.22 presents baseline versus pruned inference-time comparison during RL testing.

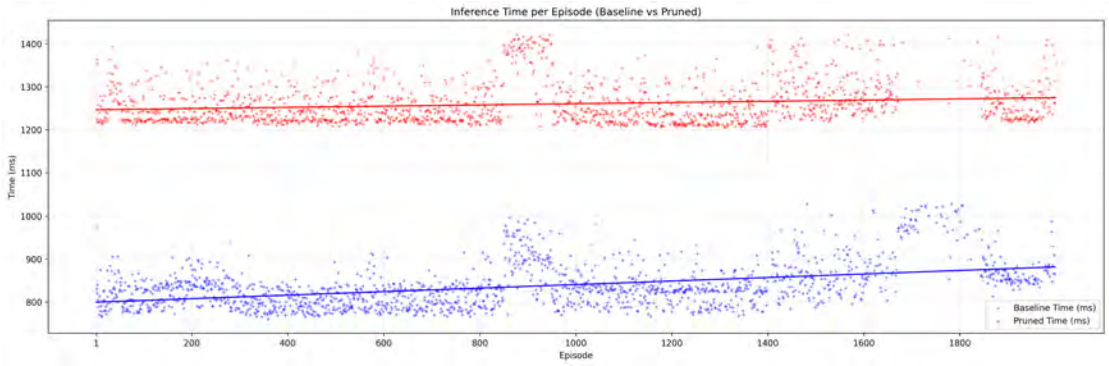


Figure 5.22: Baseline versus pruned inference-time comparison during RL testing.

Figure 5.23 presents inference-time distribution per selected action during RL testing.

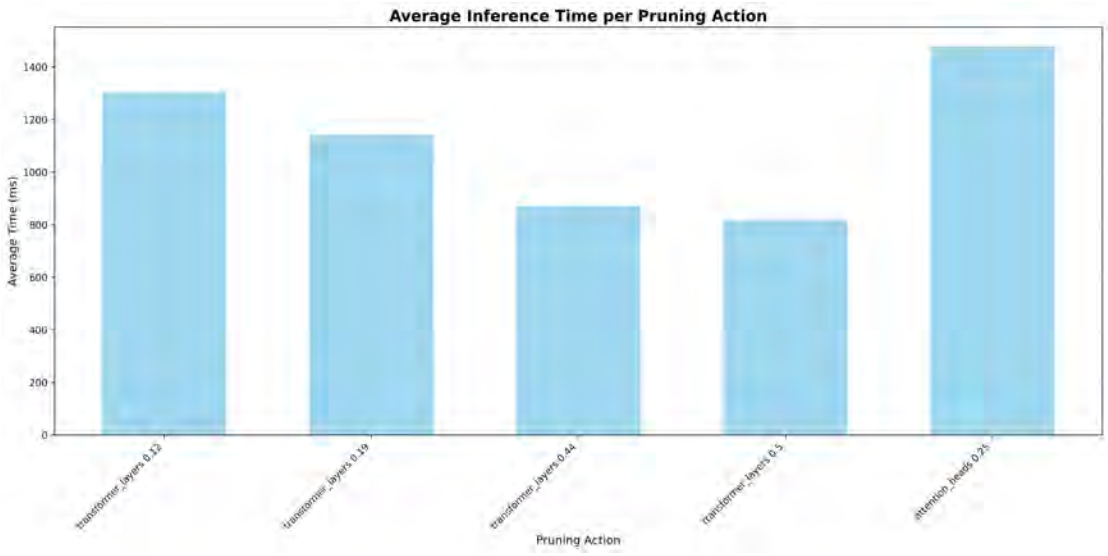


Figure 5.23: Inference-time distribution per selected action during RL testing.

The test reward progression differs sharply from the training progression. During testing, the vast majority of rewards fall in the positive range, with only occasional negative spikes corresponding to prompts for which even moderate pruning causes significant quality degradation. The absence of the large negative deviations seen during training confirms that the $\epsilon = 0$ policy avoids the destructive actions encountered during exploration. Figure 5.24 presents baseline versus pruned perplexity during RL testing.

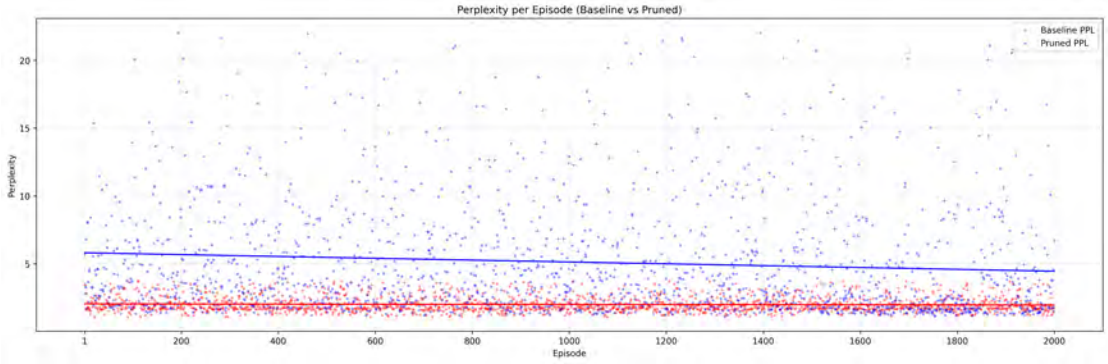


Figure 5.24: Baseline versus pruned perplexity during RL testing.

Figure 5.25 presents average perplexity per action during RL testing.

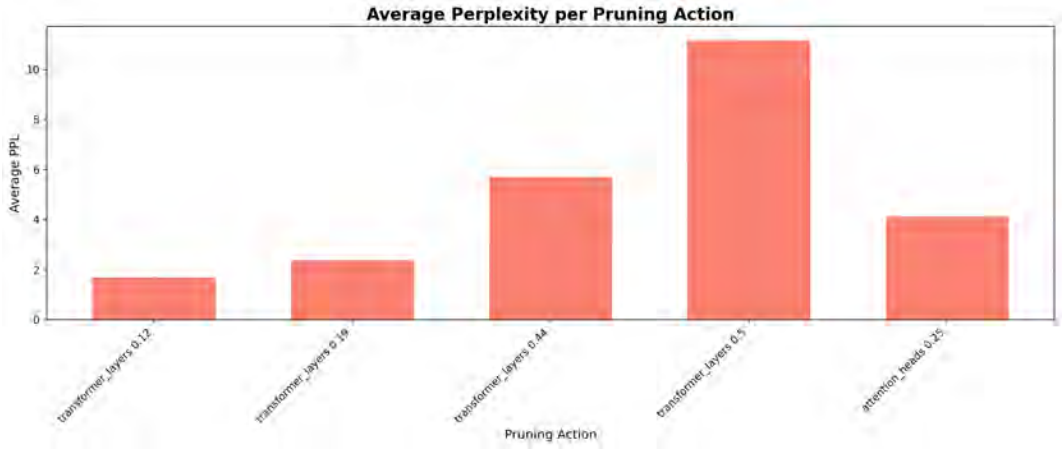


Figure 5.25: Average perplexity per action during RL testing.

Figure 5.26 presents quality-versus-speed trade-off during RL testing.

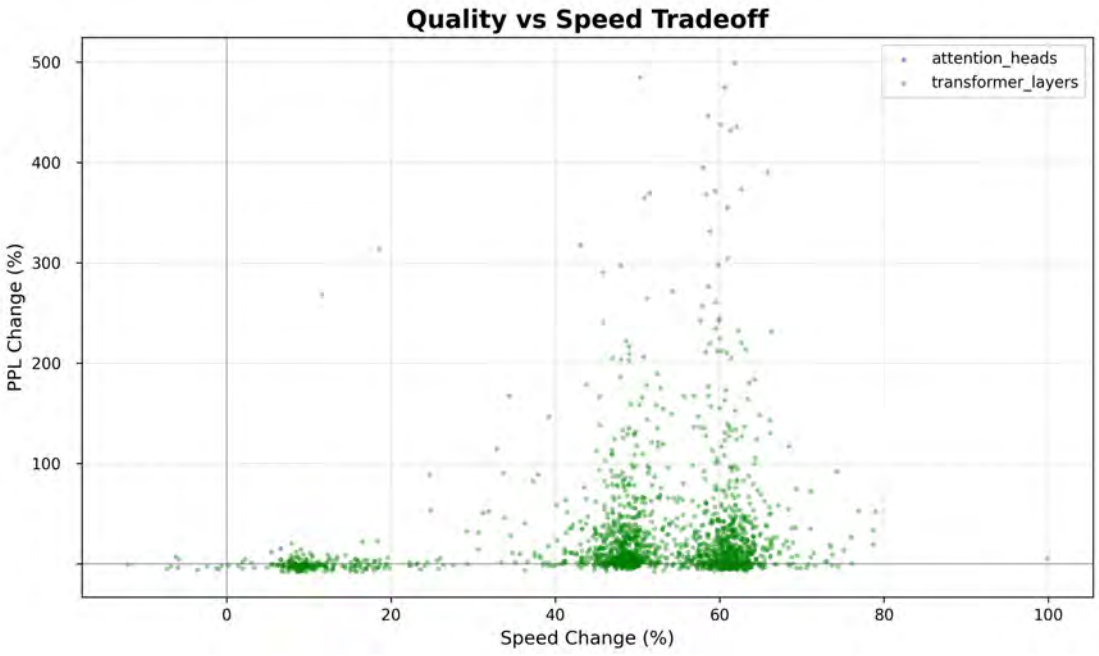


Figure 5.26: Quality-versus-speed trade-off during RL testing.

Figure 5.27 presents controller overhead breakdown during RL testing.

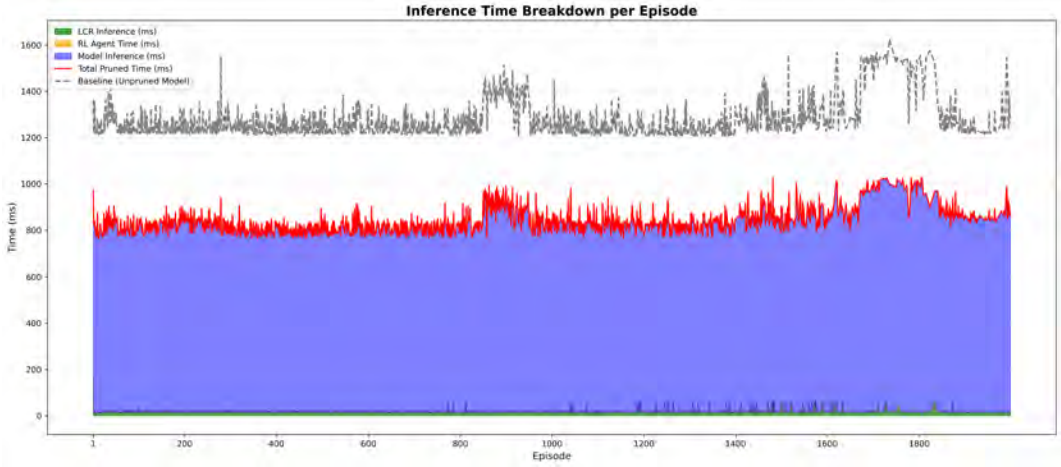
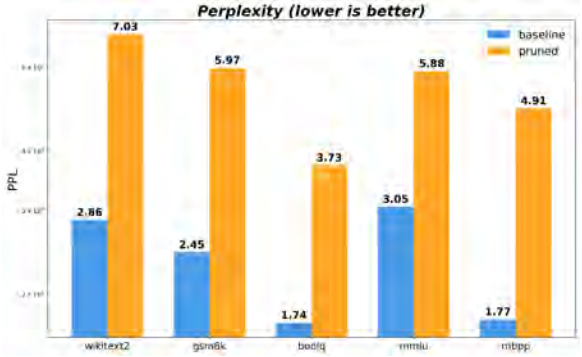
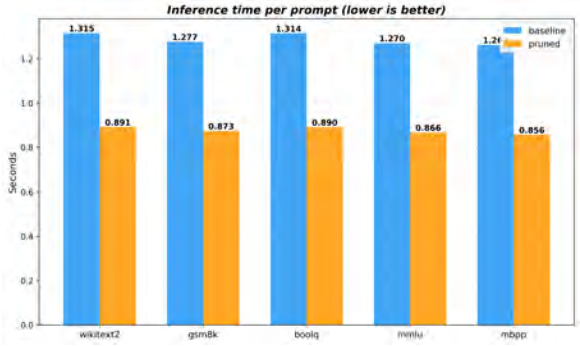


Figure 5.27: Controller overhead breakdown during RL testing.

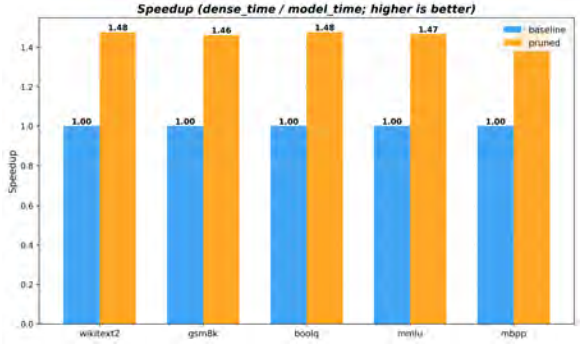
Figure 5.28 presents per-source RL testing metrics in a 22 grid. Panels (a)–(d) report perplexity, inference time, speedup, and token throughput, respectively, showing that source-level differences remain consistent under the converged exploitation policy.



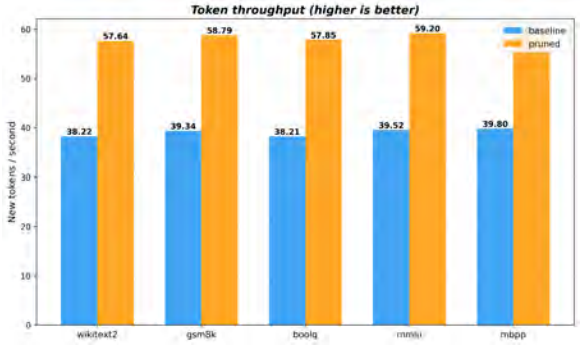
(a) Per-source perplexity.



(b) Per-source inference time.



(c) Per-source speedup.



(d) Per-source token throughput.

Figure 5.28: Per-source RL testing metrics in a 2×2 grid. Panels (a)–(d) report perplexity, inference time, speedup, and token throughput, respectively, showing that source-level differences remain consistent under the converged exploitation policy.

5.2 Analysis of Design Solutions

This section analyzes the specific design decisions made during the development of SPRINT and evaluates how each architectural choice contributes to the framework's overall performance. The analysis draws on both the quantitative results presented in Section 5.1 and the ablation studies detailed in Section 5.3.

5.2.1 Learned Prompt Sensitivity Router

The decision to replace heuristic prompt-complexity scores with a learned router trained on oracle sensitive labels is the central methodological contribution of SPRINT. The results from Section 5.1.1 validate this design choice across multiple dimensions.

Ranking Quality vs. Calibration: The Spearman $\rho = 0.797$ on the held-out test set demonstrates that the router preserves the oracle's relative sensitivity ranking with high fidelity. This is more important than absolute calibration ($R^2 = 0.633$) for the downstream RL controller because action selection depends on comparative sensitivity levels.

Domain Generalization: The per-source analysis reveals that the router generalizes across five functionally distinct domains despite being trained on a balanced mixture. The performance gradient from MBPP to GSM8K reflects the fundamental predictability structure of each domain rather than a failure of the router architecture.

Architectural Efficiency: The BERT-mini backbone adds only 17–18 ms of latency to the inference pipeline, which is negligible compared to the 400+ ms of latency saved through the resulting pruning decisions. The return on investment for the LCR is therefore substantial.

5.2.2 Reward Function Design

The normalized linear PPL reward formulation ($\alpha = 0.9$, $\beta = 0.1$, clamped to $[-1, 1]$) was designed to address the specific challenges of RL-based pruning:

1. **Proportional Quality Penalty**: Unlike a log-PPL formulation, the linear PPL ratio preserves the proportional relationship between quality degradation and penalty magnitude.

2. **Bounded Rewards**: The $[-1, 1]$ clamp prevents catastrophic pruning episodes from dominating the replay buffer's reward distribution.

3. **Speed-First Weighting**: The $\alpha = 0.9$ versus $\beta = 0.1$ ratio reflects the operational priority of the framework: bounded quality degradation is accepted in exchange for meaningful inference speedup.

5.2.3 Action Space and Policy Behavior

The action space was designed so that every discrete action maps to a mechanically distinct structural outcome. The training and test results confirm several design decisions.

Layer-Skip Dominance: The agent learned to strongly prefer layer skipping over head pruning for latency reduction. During testing, 99.95% of selections chose layer-skipping actions.

Bimodal Policy: The converged policy's concentration on two intensity levels (44% and 50%) demonstrates that the DDQN has identified the optimal operating points for the Llama 2 7B architecture.
No FFN Slicing: The absence of FFN slicing actions from the final action space is validated by the clean convergence observed during training.

5.2.4 Controller Overhead Analysis

The total controller overhead (LCR inference + RL action selection) averages **17.77 ms** during testing:
Table 5.7 presents controller overhead breakdown during testing.

Table 5.7: Controller overhead breakdown during testing.

Component	**Avg Time**	**% of Total Pruned Latency**
LCR inference	17.08 ms	1.9%
RL action selection	0.69 ms	0.08%
Total overhead	**17.77 ms**	**2.0%**

This overhead is included in all reported pruned latency figures, ensuring honest accounting. The overhead represents approximately 2.0% of the total pruned inference time, which is negligible compared to the 32% speedup achieved.

5.3 Final Design Adjustment

5.3.1 Reward Function Ablation (Study 1)

The reward-function ablation sweep evaluated five normalized (α, β) pairs where $\alpha + \beta = 1.0$, ensuring that every configuration allocates the full weight budget between speed and quality without confounded comparisons. Each configuration trained a fresh DDQN for 200 episodes on the same prompt set, enabling controlled comparison.
Table 5.8 presents reward-function ablation sweep results.

Table 5.8: Reward-function ablation sweep results.

α	β	**Avg Reward**	**Tail-20 Reward**	**Avg PPL**	**Tail-20 PPL**	**Speedup (%)**	**Tail-20 Speedup (%)**
0.9	**0.1**	−0.015	**+0.016**	12.17	12.44	**27.37**	**30.75**
0.8	0.2	−0.133	−0.078	10.35	5.08	23.64	18.04
0.7	0.3	−0.158	−0.063	7.89	2.79	20.07	9.61
0.6	0.4	−0.183	−0.033	8.71	2.50	20.64	15.21
0.5	0.5	−0.210	−0.123	13.98	50.25	19.34	10.04

Figure 5.29 presents reward-function ablation fused heatmap.

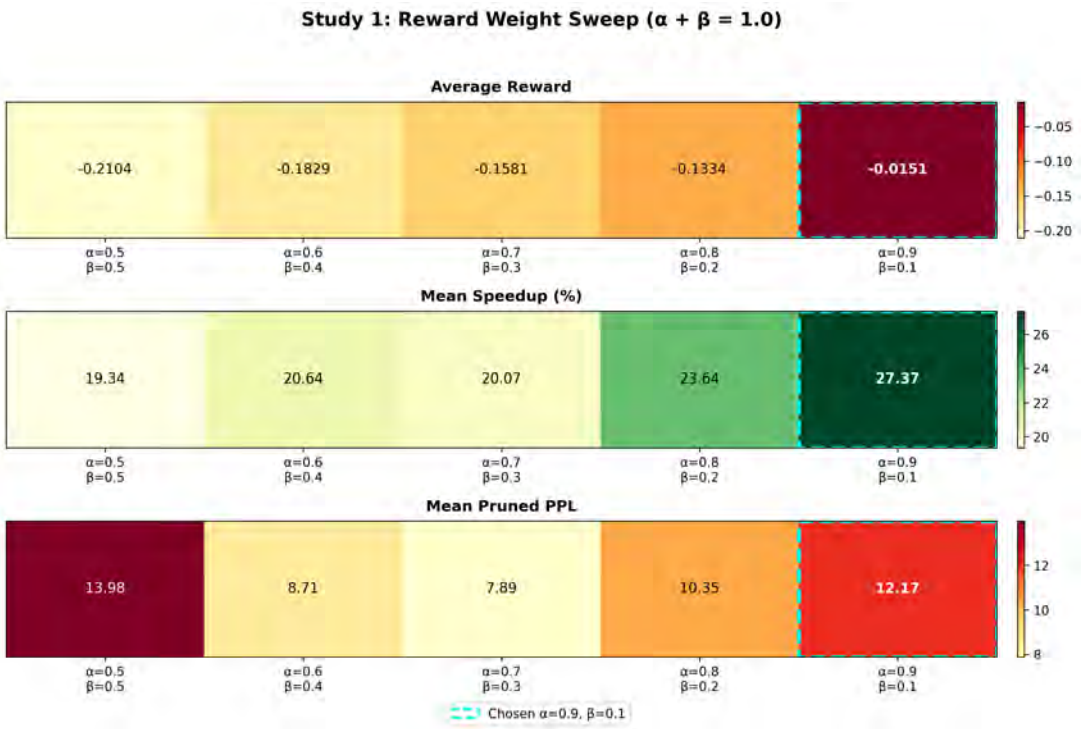


Figure 5.29: Reward-function ablation fused heatmap.

The fused heatmap provides a compact summary of the sweep results. Reading from lower α to higher α, the average reward and speedup increase monotonically, while the perplexity response is non-monotonic because the learned policy changes as the reward landscape changes.
Figure 5.30 presents radar comparison of reward-function ablation configurations.

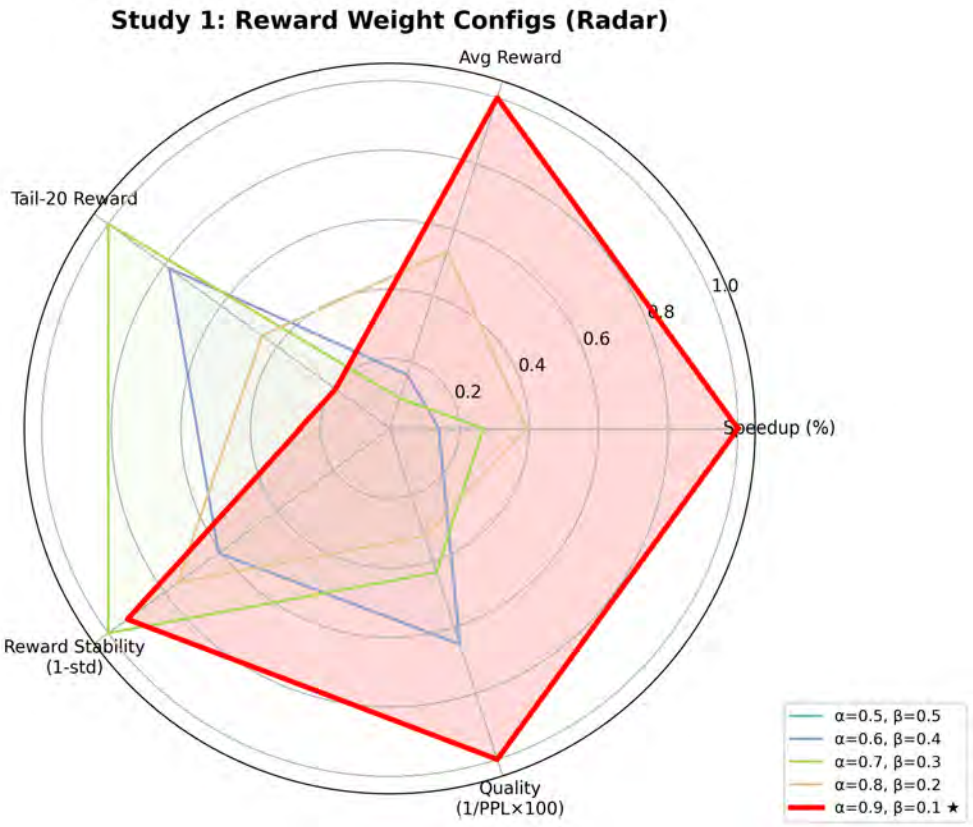


Figure 5.30: Radar comparison of reward-function ablation configurations.

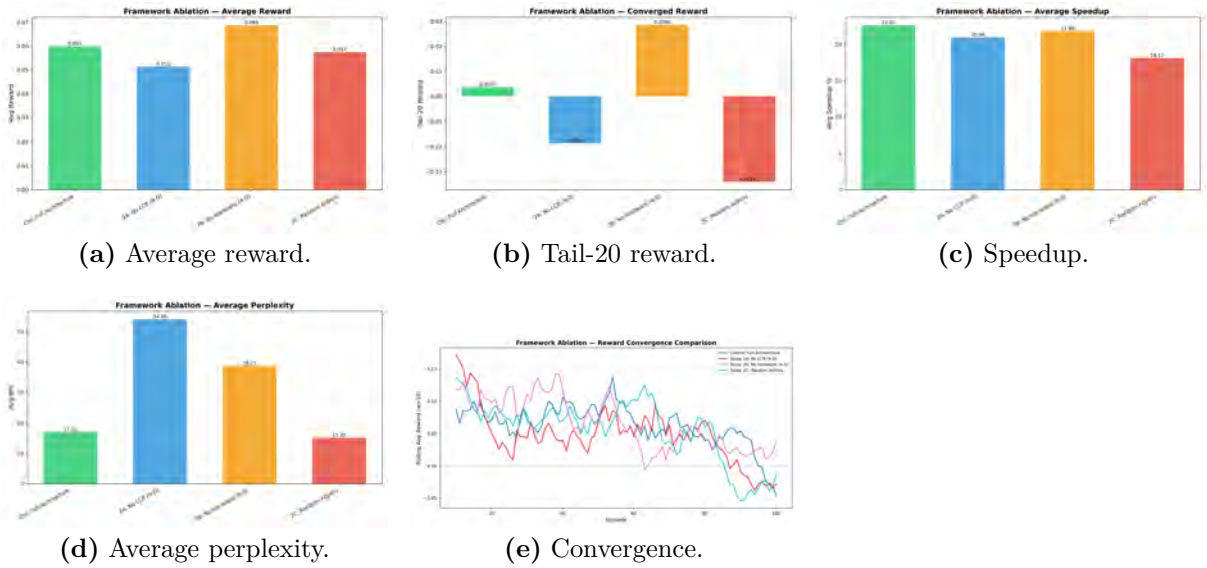
The $\alpha = 0.9, \beta = 0.1$ configuration is the clearest optimum. It is the only configuration with a positive tail-20 reward, meaning that it is the only one that yields a consistently net-positive policy in the late exploitation phase. This result justifies the final choice of a speed-first reward weighting.

5.3.2 Framework Ablation Studies (Studies 2A–2C)

The framework ablation studies isolate the contribution of each architectural component by systematically removing or replacing it while keeping all other components fixed. All experiments use interleaved execution with live baselines to eliminate systematic timing bias.

The following grid compares the four experimental conditions:

Figure 5.31 presents framework ablation study outcomes in a 23 grid. Panels (a)–(e) report average reward, tail-20 reward, speedup, average perplexity, and convergence for the full controller and the ablated variants.



(a) Average reward. **(b)** Tail-20 reward. **(c)** Speedup.

(d) Average perplexity. **(e)** Convergence.

Figure 5.31: Framework ablation study outcomes in a 2×3 grid. Panels (a)–(e) report average reward, tail-20 reward, speedup, average perplexity, and convergence for the full controller and the ablated variants.

Within this grid, the convergence panel is the most informative visualization for the ablation studies. It shows the episode-by-episode reward trajectory for all four conditions, enabling direct visual comparison of convergence speed, steady-state reward, and policy variance.

Study 2A — No LCR (9-D State): Removing the LCR sensitivity score forces the DDQN to make pruning decisions without direct prompt-level sensitivity information. The No-LCR variant still learns partially effective behaviour, but its converged reward is weaker than the full control, indicating that the LCR contributes information that cannot be fully recovered from telemetry and early-Llama features alone.

Study 2B — No Hardware (4-D State): Removing hardware telemetry leaves the agent with only the LCR score and early-Llama features. Under the controlled hardware conditions of these experiments, this ablation performs similarly to the full control because the deployment environment is stable. This should not be interpreted as evidence that hardware features are unnecessary in practice.

Study 2C — Random Actions (Baseline): Replacing the DDQN policy with random action selection provides a floor against which the learned policy can be compared. The random baseline still produces some speedup because some random layer-skipping actions reduce latency, but its tail behaviour is much worse because it wastes episodes on destructive or zero-benefit actions.

5.3.3 Final System Design

The final SPRINT system integrates three primary components:

(1) the Learned Complexity Router (LCR)
(2) the DDQN-based Adaptive Pruning Controller
(3) the Structural Pruning engine.

These components operate sequentially during inference, where each input prompt is first evaluated for sensitivity, followed by dynamic pruning decision-making based on both

prompt characteristics and system state. The pruning engine then applies the selected configuration, resulting in optimized inference execution.
The final system represents the outcome of iterative refinement and experimental validation, ensuring both performance gains and stability across diverse prompt domains.

5.4 Statistical Analysis

5.4.1 LCR Confidence Intervals and Robustness

The 95% bootstrap confidence interval for the test Spearman ρ is [**0.7787, 0.8167**], computed via 1,000 resamples of the 2,000-sample test set. The narrow width of this interval provides strong statistical assurance that the observed $\rho = 0.797$ is not an artefact of the specific test-set composition. The interval does not contain zero or even moderate correlation values, confirming that the LCR provides a statistically strong ranking signal.

5.4.2 Cross-Method Sensitivity Correlation

An important empirical finding from the oracle-labeling stage validates the multi-method composite label design:
Table 5.9 presents cross-method sensitivity correlation between head pruning and layer skipping.

Table 5.9: Cross-method sensitivity correlation between head pruning and layer skipping.

Metric	**Head pruning vs. layer skipping**
Spearman ρ	≈ 0.172
Pearson r	≈ 0.214
R^2	≈ 0.046

Head-pruning sensitivity and layer-skipping sensitivity are only weakly correlated, indicating that less than 5% of the variance in one type of sensitivity is explained by the other. This weak relationship is expected because the two operators stress different parts of the model: head pruning reduces attention diversity, whereas layer skipping reduces feature-processing depth. This result supports the use of multi-method oracle labeling and justifies treating different pruning operators as distinct intervention modes rather than as interchangeable forms of generic difficulty.

5.4.3 Per-Source Statistical Breakdown

The per-source metrics reveal statistically significant differences in the LCR's predictive accuracy across benchmark domains:
Table 5.10 presents per-source statistical breakdown of LCR test performance.

Table 5.10: Per-source statistical breakdown of LCR test performance.

Source	**Test ρ**	**Test R^2**	**Test MSE**	**Interpretation**
MBPP	0.886	0.794	0.013	Strong structural cues; best predictability
BoolQ	0.796	0.622	0.023	Consistent passage+question format; good predictability
MMLU	0.766	0.640	0.032	Structured multiple-choice format with domain diversity
WikiText-2	0.625	0.373	0.038	High lexical redundancy obscures sensitivity signals
GSM8K	0.561	0.257	0.034	Sensitivity driven by backbone-internal reasoning; lowest predictability

The MBPP-to-GSM8K performance gradient quantifies the range of domain-dependent predictability in SPRINT. Code generation is the easiest to model because syntax, delimiters, and structural markers provide strong signals. Mathematical reasoning is hardest because sensitivity depends on the internal reasoning trajectory of the backbone rather than on visible surface structure alone.

5.4.4 Quality-Speed Pareto Analysis

The quality-vs-speed scatter plots from the RL test phase reveal the Pareto frontier of achievable trade-offs under the learned policy. The Pareto-optimal points represent configurations where no improvement in speed is possible without sacrificing quality, and vice versa.

The key observation from the Pareto analysis is that the converged policy's two preferred actions, 44% and 50% layer skip, occupy near-optimal positions on the frontier. The 44% action achieves approximately 32.4% speedup at moderate quality cost, whereas the 50% action pushes further toward speed with higher PPL. Together, these define the practical operating range of the framework for this backbone.

5.5 Comparisons and Relationships

This section presents head-to-head comparisons between SPRINT and three established static pruning baselines from the literature: SparseGPT, Wanda, and LLM Pruner. All comparison experiments were conducted on a consistent evaluation protocol. While the primary SPRINT experiments in Sections 5.1–5.4 use the Llama 2 7B backbone on the RTX 5090, the comparison results here are obtained from corresponding comparison runs against these static baselines to establish the relative strengths and weaknesses of each approach.

5.5.1 Comparison with SparseGPT

SparseGPT is a one-shot weight-pruning method that uses approximate second-order information to prune weight matrices to a target sparsity while minimizing layer-wise reconstruction error. Two configurations were evaluated.

SparseGPT 50% Unstructured Sparsity

Table 5.11 presents sparseGPT 50% unstructured sparsity results.

Table 5.11: SparseGPT 50% unstructured sparsity results.

Benchmark	**Baseline PPL**	**Pruned PPL**	**PPL Ratio**	**Baseline tok/s**	**Pruned tok/s**	**Speedup**
WikiText-2	12.68	20.40	1.61×	26.22	27.47	1.07×
GSM8K	11.55	14.17	1.23×	28.14	28.29	1.01×
BoolQ	12.73	22.05	1.73×	27.95	25.83	0.82×
MMLU	9.03	14.28	1.58×	24.38	28.52	0.71×
MBPP	19.98	39.14	1.96×	26.90	25.34	0.85×

Zero-shot Accuracy (200 samples): BoolQ 59.5% → 56.0% (−3.5 pp), MMLU 40.5% → 26.5% (−14.0 pp).
SparseGPT 50% unstructured sparsity produces moderate quality degradation but virtually no inference speedup. On several benchmarks, the pruned model is actually slower than the baseline. This occurs because unstructured sparsity creates irregular sparsity patterns that cannot be efficiently exploited by standard GPU hardware. This highlights a critical advantage of SPRINT's structural pruning approach: physical layer removal produces genuine latency reductions.

SparseGPT 2:4 Semi-Structured Sparsity

Table 5.12 presents sparseGPT 2:4 semi-structured sparsity results.

Table 5.12: SparseGPT 2:4 semi-structured sparsity results.

Benchmark	**Baseline PPL**	**Pruned PPL**	**PPL Ratio**	**Speedup**
WikiText-2	12.68	31.39	2.48×	0.86×
GSM8K	11.55	20.44	1.77×	0.88×
BoolQ	12.73	39.30	3.09×	0.79×
MMLU	9.03	24.32	2.69×	0.13×
MBPP	19.98	53.53	2.68×	0.94×

Zero-shot Accuracy: BoolQ 59.5% → 47.5% (−12.0 pp), MMLU 40.5% → 9.0% (−31.5 pp).
This configuration causes significantly worse quality degradation than the unstructured variant while providing even less speedup. The 2:4 pattern therefore performs poorly in the tested setting.

5.5.2 Comparison with Wanda

Wanda is an activation-aware weight-pruning method. Three full sparsity configurations were evaluated.

Wanda 50% Unstructured

Table 5.13 presents wanda 50% unstructured results.

Table 5.13: Wanda 50% unstructured results.

Benchmark	Baseline PPL	Pruned PPL	PPL Ratio	Speedup
WikiText-2	13.57	33.59	2.48×	0.92×
GSM8K	4.57	7.48	1.64×	1.12×
BoolQ	14.62	15.88	1.09×	1.28×
MMLU	3.57	4.37	1.22×	0.18×
MBPP	3.83	7.02	1.83×	1.17×

Zero-shot Accuracy: BoolQ 60.5% → 60.0% (−0.5 pp), MMLU 47.0% → 27.5% (−19.5 pp).

Wanda 50% with 2:4 Structure

Table 5.14 presents wanda 50% with 2:4 structure results.

Table 5.14: Wanda 50% with 2:4 structure results.

Benchmark	Baseline PPL	Pruned PPL	PPL Ratio	Speedup
WikiText-2	13.57	106.94	7.88×	1.14×
GSM8K	4.57	19.45	4.26×	1.03×
BoolQ	14.62	32.13	2.20×	0.93×
MMLU	3.57	41.12	11.51×	0.12×
MBPP	3.83	32.11	8.39×	1.06×

Zero-shot Accuracy: BoolQ 60.5% → 56.5% (−4.0 pp), MMLU 47.0% → 24.0% (−23.0 pp).

Wanda 50% with 4:8 Structure

Table 5.15 presents wanda 50% with 4:8 structure results.

Table 5.15: Wanda 50% with 4:8 structure results.

Benchmark	Baseline PPL	Pruned PPL	PPL Ratio	Speedup
WikiText-2	13.57	60.79	4.48×	1.10×
GSM8K	4.57	12.22	2.68×	1.04×
BoolQ	14.62	16.29	1.11×	0.98×
MMLU	3.57	6.20	1.74×	0.20×
MBPP	3.83	11.46	2.99×	0.97×

Zero-shot Accuracy: BoolQ 60.5% → 59.0% (−1.5 pp), MMLU 47.0% → 20.0% (−27.0 pp).
Across all Wanda configurations, the same broad pattern appears: weight sparsity does not reliably convert into latency savings on the tested hardware, while quality degradation grows quickly as the sparsity structure becomes more aggressive.

5.5.3 Comparison with LLM Pruner

LLM Pruner is a structured pruning method that removes groups of coupled structures and can optionally apply LoRA recovery.

LLM Pruner 25% (without LoRA recovery)

Table 5.16 presents lLM Pruner 25% without LoRA recovery.

Table 5.16: LLM Pruner 25% without LoRA recovery.

Benchmark	Baseline PPL	Pruned PPL	PPL Ratio	Speedup
WikiText-2	17.71	52.83	2.98×	0.99×
GSM8K	4.61	27.42	5.95×	3.28×
BoolQ	15.36	71.42	4.65×	1.12×
MMLU	3.50	74.82	21.37×	0.34×
MBPP	3.43	12.78	3.72×	0.99×

LLM Pruner 25% (with LoRA Recovery)

Table 5.17 presents lLM Pruner 25% with LoRA recovery.

Table 5.17: LLM Pruner 25% with LoRA recovery.

Benchmark	Baseline PPL	Pruned PPL	LoRA PPL	LoRA PPL Ratio	Speedup (LoRA)
WikiText-2	17.71	52.83	40.71	2.30×	0.62×
GSM8K	4.61	27.42	12.12	2.63×	2.50×
BoolQ	15.36	71.42	37.58	2.45×	1.16×
MMLU	3.50	74.82	14.04	4.01×	0.61×
MBPP	3.43	12.78	7.23	2.11×	0.55×

Zero-shot Accuracy (LoRA): BoolQ 67.0% → 54.5% (pruned) → 54.5% (pruned+LoRA), MMLU 33.0% → 19.5%.
LLM Pruner demonstrates the core trade-off of structured pruning with recovery. Without LoRA, quality degradation is severe. With LoRA, some quality is recovered, but the LoRA adapter introduces runtime overhead that cancels much of the speed advantage. This differs substantially from SPRINT, which preserves reversibility without adding a permanent recovery module to the inference path.

5.5.4 Unified Cross-Method Comparison

The following two tables provide a unified comparison of all evaluated methods. The first focuses on perplexity, inference time, token throughput, and speedup, while the second reports zero-shot accuracy:
Table 5.18 presents comparison of pruning methods on perplexity, inference time, token throughput, and speedup. Values are reported as . Sprint is highlighted in bold.

Table 5.18: Comparison of pruning methods on perplexity, inference time, token throughput, and speedup. Values are reported as *before pruning* → *after pruning*. Sprint is highlighted in bold.

Paper	**Pruning type**	**Perplexity ↓**	**Inference time / prompt ↓ (s)**	**Token throughput ↑ (tok/s)**	**Speedup ↑**
LLMPruner	Single reported setting	8.92 → 47.85	1.05 → 1.00	38.90 → 37.59	1.00 → 1.28
SparseGPT	2:4 structured	13.19 → 33.79	1.34 → 1.85	27.62 → 26.80	1.00 → 0.72
SparseGPT	50% unstructured	13.19 → 22.01	1.37 → 1.49	26.72 → 27.09	1.00 → 0.89
Wanda	2:4 structured	8.03 → 46.35	1.36 → 1.59	30.49 → 30.68	1.00 → 0.86
Wanda	4:8 structured	8.03 → 21.39	1.31 → 1.42	31.79 → 31.89	1.00 → 0.86
Wanda	Unstructured	8.03 → 13.67	1.54 → 1.60	26.42 → 30.59	1.00 → 0.94
Sprint	**Head pruning and Layer skipping**	**2.30 → 3.59**	**1.61 → 1.32**	**32.99 → 41.71**	**1.00 → 1.23**

Table 5.19 presents zero-shot accuracy comparison. Values are reported as . Sprint is highlighted in bold.

Table 5.19: Zero-shot accuracy comparison. Values are reported as *before pruning* → *after pruning*. Sprint is highlighted in bold.

Paper	**Pruning type**	**BoolQ accuracy (%)**	**MMLU accuracy (%)**
LLMPruner	Single reported setting	67.0 → 48.5	33.0 → 19.5
SparseGPT	2:4 structured	59.5 → 47.5	40.5 → 9.0
SparseGPT	50% unstructured	59.5 → 56.0	40.5 → 26.5
Wanda	2:4 structured	60.5 → 59.0	47.0 → 20.0
Wanda	4:8 structured	60.5 → 60.0	47.0 → 27.5
Wanda	Unstructured	60.5 → 56.5	47.0 → 24.0
Sprint	**Head pruning and Layer skipping**	**59.5 → 54.5**	**47.5 → 38.5**

Several points emerge from these cross-method comparisons. First, SPRINT is the only method that achieves both substantial speedup and runtime adaptivity. Second, the static pruning baselines either fail to deliver real speedup or suffer large quality penalties. Third, SPRINT does not require calibration inference for pruning-time importance ranking in the same sense as several static baselines. Fourth, SPRINT preserves reversibility and can vary its pruning decision per prompt, which static methods cannot do.

5.6 Discussion

5.6.1 Summary of Results

The five-stage SPRINT pipeline produced results that satisfy the functional requirements specified in Chapter 3 under the evaluated hardware and workload conditions. The Learned Complexity Router converged at epoch 33 of 50, reaching a test Spearman correlation of $\rho = 0.797$ with a 95% bootstrap confidence interval of $[0.779, 0.817]$, and achieving $R^2 = 0.633$. Both measures exceed the FR-3 threshold ($\rho \geq 0.70$), and the margin above the threshold held across all five benchmark sources.

The DDQN controller, trained over 8,000 episodes with exponential epsilon decay from 1.0 to 0.10, converged to a policy that achieved a 32.0% average inference speedup across 2,000 held-out test episodes. Average baseline latency was 1,287.61 ms and average pruned latency was 875.39 ms. Total controller overhead was 17.77 ms, corresponding to approximately 2.0% of total latency. This overhead includes both the LCR routing stage and the DDQN action selection step, and remains small relative to the recovered speedup on the RTX 5090 evaluation system.

The oracle analysis also showed that head-pruning sensitivity and layer-skipping sensitivity are only weakly correlated across the 10,000-prompt dataset ($\rho \approx 0.172$, $R^2 \approx 0.046$). This result provides an empirical basis for treating the two operators as distinct intervention modes rather than collapsing sensitivity into a single measure. A prompt that tolerates narrower attention may still degrade under shallower computation, and the oracle labeling captures these failure modes as separate signals.

The ablation results quantify the contribution of each pipeline component. Removing the LCR score reduced converged policy quality relative to the full 10-dimensional control signal. Removing hardware telemetry produced similar performance under the stable test conditions used here, though the interpretation of this result is limited to the evaluated environment. By contrast, replacing the DDQN with uniform random action selection (Study 2C) reduced tail-20 speedup from 17.67% to 4.79%, confirming that the learned policy selects actions that random selection does not reproduce consistently.

5.6.2 Analysis of Key Design Decisions

The choice of physical layer removal rather than identity-forward monkey patching was the most consequential engineering decision in the project. Early experiments using monkey patching produced pruned inference that ran 5–15% slower than the dense baseline. The root cause was a cache-alignment failure: skipped layers did not execute the cache update routine, causing Hugging Face DynamicCache to misalign subsequent layer reads against stale key–value entries. Physical removal from the module list, followed by sequential reassignment of `layer_idx`, removed this misalignment and converted negative speedups into the positive gains reported at test time.

For the router architecture, the progression from a bag-of-words MLP ($\rho < 0.45$) through TinyBERT and DistilBERT to BERT-mini reflects a trade-off between routing latency and ranking quality. The router must remain within a tight overhead budget; if routing consumes too much time, it reduces or eliminates the net benefit of pruning. ScalarMix pooling over all hidden states was selected over CLS-only extraction because pruning sensitivity relates to distributed prompt properties rather than a single pooled representation associated with BERT's pretraining objectives (Peters et al., 2018; Reimers and Gurevych, 2019).

Reward-function ablation confirmed that the $(\alpha = 0.9, \beta = 0.1)$ configuration was the only setting in the tested sweep that yielded a positive tail-20 reward. Configurations that weighted quality more heavily converged to weaker policies because the penalty dominated the learning signal before the agent accumulated enough experience to distinguish moderate degradation from catastrophic failures. The linear perplexity ratio formulation, clamped to $[-1, 1]$, preserves proportionality—e.g., a $10\times$ perplexity increase produces approximately ten times the penalty of a $1\times$ increase—while preventing any single extreme episode from distorting replay-buffer learning.

The GQA-safe head-pruning constraint is necessary for correctness. Llama-2-7B uses grouped attention structure that requires pruning to respect head-group wiring. The pruning engine therefore removes mechanically valid head groups at each intensity level, preventing structural mismatches that would otherwise introduce instability and large perplexity spikes.

5.6.3 Limitations

The R^2 ceiling of 0.633 implies that roughly 37% of the variance in oracle pruning sensitivity cannot be predicted from prompt text alone. This residual variance is plausibly attributable to backbone-internal dynamics—activation magnitudes, residual-stream norms, and gating patterns—that are not observable without running the full model forward pass. This ceiling is therefore an empirical property of the prediction task rather than only a limitation of the router architecture. In practice, the controller will occasionally misclassify prompt sensitivity and apply pruning that incurs larger quality loss than expected.

Zero-shot accuracy drops were larger than expected on BoolQ, where the dense-to-pruned gap reached 17 percentage points (62.0% to 45.0%). The MMLU gap was smaller at 8.8 percentage points (34.3% to 25.5%). These figures suggest that the current reward function does not penalise discrete accuracy loss directly, and instead uses perplexity ratio as a proxy for quality. A reward term that incorporates downstream task accuracy would likely shift the learned policy toward more conservative pruning on classification-sensitive prompts.

All experiments were conducted on a single hardware configuration: an RTX 5090 GPU paired with the evaluated CPU and memory configuration. The converged action distribution reflects that specific latency surface. Deploying the same trained controller on substantially different hardware—particularly CPU-only environments or GPUs with different memory-bandwidth characteristics—would likely require retraining using telemetry sampled from the target platform.

The current implementation does not support batched inference. Each episode processes a single prompt, which matches interactive local deployment but does not cover server-side settings where multiple requests share GPU resources simultaneously. Batched operation would require revising both the reward definition and the telemetry interpretation, since per-prompt latency depends on concurrent load.

5.6.4 Relationship to Prior Work

SparseGPT [9] and Wanda [13] apply weight sparsity at a fixed ratio before deployment. In the comparison experiments, their main limitation is that unstructured sparsity produces irregular weight patterns that standard dense CUDA kernels do not exploit efficiently. As a result, parameter reduction does not consistently translate into latency reduction on consumer GPU hardware. SPRINT's physical layer removal avoids this mismatch by operating at the granularity of whole transformer blocks, which align directly with the forward-pass computation graph and yield predictable latency savings without requiring sparse-kernel support.

LLM-Pruner [4] achieves structural compression through grouped neuron removal and recovers quality through a LoRA fine-tuning step. While effective for restoring quality, the adapter is permanently attached to the inference path and removes the per-episode reversibility required for SPRINT's runtime adaptation.

RAP [5] is the closest prior method in intent, using reinforcement learning to select pruning configurations at runtime. However, its routing signal is derived from heuristic proxies rather than measured degradation on the target backbone. In SPRINT, oracle-derived labels measure observed loss gaps under the same operators and model used at deployment, giving the sensitivity signal direct operational meaning.

Chapter 6
Conclusion

6.1 Conclusion

6.1.1 Summary of Findings

This work addressed one question: can structural pruning intensity be selected per prompt at inference time in a way that yields measurable speedup without relying on a fixed offline compression profile? Based on the experiments reported in Chapter 5, the answer is yes, with explicit trade-offs in downstream quality and with dependence on the evaluated hardware context.

The pipeline integrates five stages. A 10,000-prompt dataset drawn equally from GSM8K (Cobbe et al., 2021), MBPP (Austin et al., 2021), WikiText-2 (Merity et al., 2017), MMLU (Hendrycks et al., 2021), and BoolQ (Clark et al., 2019) supported oracle labeling across varied prompt types. The oracle measures non-negative loss gaps under fixed pruning intensities and forms a composite sensitivity score from operator-specific degradation measures. This approach defines sensitivity by measured behaviour on the target backbone rather than by surface proxies such as length or token statistics.

The Learned Complexity Router—BERT-mini fine-tuned with ScalarMix pooling, attention-statistics extraction, and auxiliary text features—predicts these oracle labels with test performance of $\rho = 0.797$ and $R^2 = 0.633$. The DDQN controller, trained over 8,000 episodes using a 10-dimensional state vector combining telemetry, router score, and early backbone signals, converged to a policy achieving a 32.0% average inference speedup across 2,000 held-out test episodes. Average baseline latency was 1,287.61 ms and average pruned latency was 875.39 ms, while total overhead across routing and action selection averaged 17.77 ms (approximately 2.0% of total latency).

Comparison experiments against SparseGPT, Wanda, and LLM Pruner indicate that reduced parameter count via unstructured sparsity does not reliably translate into latency reduction on standard dense GPU kernels. SPRINT's structural approach—physical removal of transformer layers—avoids this kernel mismatch and yields speedups that scale more predictably with pruning intensity.

This work demonstrates that adaptive, runtime-aware optimization can significantly improve the practicality of deploying large language models, bridging the gap between high-performance AI systems and resource-constrained environments.

6.1.2 Contributions

Four contributions follow from this work.

1. **Oracle labeling methodology.** The oracle computes sensitivity from observed dense-versus-pruned degradation on the target backbone under each operator. The weak cross-operator correlation ($\rho \approx 0.172$) supports the conclusion that head pruning and layer skipping stress different components and should not be treated as interchangeable signals.

2. **Learned Complexity Router design.** Combining ScalarMix over BERT-mini hidden states, attention-statistics features, and lightweight auxiliary text features yields high ranking fidelity ($\rho = 0.797$) while keeping the routing cost within the practical overhead budget needed for runtime adaptation.

3. **Resolution of cache misalignment in layer skipping.** Identity-forward layer skipping can lead to DynamicCache misalignment because skipped layers do not update key–value cache entries. Physical removal from the module list, followed by sequential reassignment of `layer_idx`, corrects cache indexing and restores expected latency behaviour.

4. **Integrated adaptive inference loop.** SPRINT combines learned sensitivity prediction, live system telemetry, and reinforcement learning into a single per-prompt decision loop. The random-policy ablation provides direct evidence that the learned controller selects actions that yield better speedup behaviour than chance.

6.1.3 Limitations

Three limitations qualify the scope of the reported results.
First, the router's R^2 ceiling indicates irreducible uncertainty for text-only sensitivity prediction. This limitation implies occasional misclassification and therefore occasional over-pruning.
Second, downstream accuracy drops on BoolQ and MMLU indicate that perplexity ratio does not fully capture task correctness. The current reward function encourages speed improvements even when discrete classification accuracy is degraded.
Third, the results were obtained on a single platform centred on an RTX 5090 GPU. The learned policy reflects that hardware's latency surface and does not guarantee transfer to other devices without retraining.

6.1.4 Future Work

The weak correlation between operator sensitivities motivates training separate routers for head pruning and layer skipping and providing operator-specific sensitivity signals to the controller. This would preserve information that is averaged away by a composite label.
Reward design is another direction with clear impact. Incorporating a small accuracy-based term would better align learning with tasks where correctness matters more than generation fluency, at the cost of additional per-episode measurement overhead.
Extending the pruning engine to support joint actions (simultaneous head pruning and layer skipping) could expose a richer quality–speed frontier than either operator reaches alone.
Finally, cross-hardware transfer remains open. The telemetry features provide a mechanism for adaptation, but whether a policy trained on an RTX 5090 transfers to CPU-only or mobile environments without retraining is untested. Establishing transfer performance, or developing a lightweight recalibration procedure for new devices, would broaden deployment applicability.

Bibliography

[1] X. Zhu, J. Li, Y. Liu, C. Ma, and W. Wang, "A survey on model compression for large language models," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 1556-1577, 2024.

[2] G. Bai, Z. Chai, C. Ling, et al., "Beyond efficiency: A systematic survey of resource-efficient large language models," *arXiv preprint arXiv:2401.00625*, 2024.

[3] Y. Jiang, H. Wang, L. Xie, H. Zhao, C. Zhang, H. Qian, and J. C.S. Lui, "D-llm: A token adaptive computing resource allocation strategy for large language models," in *38th Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[4] X. Ma, G. Fang, and X. Wang, "LLM-Pruner: On the structural pruning of large language models," *arXiv preprint arXiv:2305.11627*, 2023.

[5] H. Liu, C. Tian, X. Wei, J. Dai, Q. Liu, T. Wei, Q. Li, and L. Li, "RAP: Runtime-adaptive pruning for llm inference," *arXiv preprint arXiv:2505.17138*, 2025.

[6] J. Lin, J. Tang, H. Tang, et al., "AWQ: Activation-aware weight quantization for LLM compression and acceleration," *arXiv preprint arXiv:2306.00978*, 2024.

[7] M. Federici, D. Belli, M. van Baalen, et al., "Efficient LLM inference using dynamic input pruning and cache-aware masking," *arXiv preprint arXiv:2405.15286*, 2024.

[8] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "GPTQ: Accurate post-training quantization for generative pre-trained transformers," in *International Conference on Learning Representations (ICLR)*, 2023.

[9] E. Frantar and D. Alistarh, "SparseGPT: Massive language models can be accurately pruned in one-shot," in *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2023, pp. 10323–10337.

[10] G. Yang, C. He, J. Guo, et al., "LLMCBench: Benchmarking large language model compression for efficient deployment," in *38th Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[11] Y. Chen, B. Cheng, J. Han, Y. Zhang, Y. Li, and S. Zhang, "DLP: Dynamic Layer-wise Pruning in Large Language Models," *arXiv preprint arXiv:2505.23807*, 2025.

[12] Y. Liu, H. Yang, Y. Chen, R. Zhang, M. Wang, Y. Du, and L. Du, "PAT: Pruning-Aware Tuning for Large Language Models," in *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence*, vol. 39, no. 23, pp. 24686–24695, 2025.

[13] M. Sun, Z. Liu, A. Bair, and J. Z. Kolter, "A Simple and Effective Pruning Approach for Large Language Models," in *International Conference on Learning Representations (ICLR)*, 2024.

[14] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "LLaMA: Open and Efficient Foundation Language Models," *arXiv preprint arXiv:2302.13971*, 2023.

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” in *Proceedings of the 2019 NAACL-HLT*, Minneapolis, MN, USA, 2019, pp. 4171–4186.

[16] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, “Measuring Massive Multitask Language Understanding,” *arXiv preprint arXiv:2009.03300*, 2020.

[17] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, C. Hesse, and J. Schulman, “Training Verifiers to Solve Math Word Problems,” *arXiv preprint arXiv:2110.14168*, 2021.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, pp. 529–533, 2015.

[19] H. van Hasselt, A. Guez, and D. Silver, “Deep Reinforcement Learning with Double Q-learning,” in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI)*, 2016, pp. ...

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention Is All You Need,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017, pp. 5998–6008.

[21] D. Hendrycks and K. Gimpel, ”Gaussian Error Linear Units (GELUs),” *arXiv preprint arXiv:1606.08415*, 2016.

[22] M. E. Peters, M. Neumann, M. Iyyer, et al., ”Deep Contextualized Word Representations,” in *Proceedings of NAACL-HLT*, 2018.

[23] I. Tenney, D. Das, and E. Pavlick, ”BERT Rediscovers the Classical NLP Pipeline,” in *Proceedings of ACL*, 2019.

[24] K. Clark, U. Khandelwal, O. Levy, and C. D. Manning, ”What Does BERT Look At? An Analysis of BERT’s Attention,” in *BlackboxNLP Workshop, ACL*, 2019.

[25] N. Reimers and I. Gurevych, ”Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks,” in *Proceedings of EMNLP*, 2019.

[26] J. Howard and S. Ruder, ”Universal Language Model Fine-tuning for Text Classification,” in *Proceedings of ACL*, 2018.

[27] I. Turc, M.-W. Chang, K. Lee, and K. Toutanova, ”Well-Read Students Learn Better: On the Importance of Pre-training Compact Models,” *arXiv preprint arXiv:1908.08962*, 2019.

[28] P. Bhargava, A. Drozd, and A. Rogers, ”Generalization in NLI: Ways (Not) to Go Beyond Simple Heuristics,” in *EMNLP Insights Workshop*, 2021.

[29] I. Loshchilov and F. Hutter, ”SGDR: Stochastic Gradient Descent with Warm Restarts,” in *International Conference on Learning Representations (ICLR)*, 2017.

[30] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization," in *International Conference on Learning Representations (ICLR)*, 2019.

[31] P. J. Huber, "Robust Estimation of a Location Parameter," *Annals of Mathematical Statistics*, vol. 35, 1964.

[32] R. Girshick, "Fast R-CNN," in *Proceedings of ICCV*, 2015.

[33] C. Sun, X. Qiu, Y. Xu, and X. Huang, "How to Fine-Tune BERT for Text Classification," in *CCL: Chinese Computational Linguistics*, 2019.

[34] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," in *Proceedings of CVPR*, 2016.

[35] R. Müller, S. Kornblith, and G. Hinton, "When Does Label Smoothing Help?," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[36] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time Analysis of the Multiarmed Bandit Problem," *Machine Learning*, vol. 47, 2002.

[37] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction," 2nd ed., MIT Press, 2018.

[38] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning Both Weights and Connections for Efficient Neural Networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

[39] R. Pope, S. Douglas, A. Chowdhery, et al., "Efficiently Scaling Transformer Inference," in *Proceedings of Machine Learning and Systems (MLSys)*, 2023.

[40] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, vol. 27, 1948.

[41] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," *Journal of Artificial Intelligence Research*, vol. 16, 2002.

[42] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Regularizing and Optimizing LSTM Language Models," in *International Conference on Learning Representations (ICLR)*, 2018.

[43] G. Melis, C. Dyer, and P. Blunsom, "On the State of the Art of Evaluation in Neural Language Models," in *International Conference on Learning Representations (ICLR)*, 2018.

[44] T. Wolf, L. Debut, V. Sanh, et al., "Transformers: State-of-the-Art Natural Language Processing," in *Proceedings of EMNLP (System Demonstrations)*, 2020.

[45] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," *arXiv preprint arXiv:1503.02531*, 2015.