Runtime Compute Control: Explicit Inference-Time Computation Management for Large Language Models

Jesús Tabares Montilla
Independent Researcher
jesus@tabares.eu
https://tabares.eu

Abstract

We introduce Runtime Compute Control (RCC), a system-level approach for explicitly managing computation during LLM inference without modifying the underlying model weights. Unlike compression techniques (quantization, pruning) that optimize storage, or architectural changes (MoE, early exit) that require training, RCC operates as a post-training execution layer that selectively controls which computational paths are evaluated at runtime.

RCC exposes a single control parameter—the compute ratio—that determines the fraction of feed-forward network computation executed per forward pass. The system maintains the original model checkpoint unmodified; computation reduction occurs entirely at the execution level.

We validate RCC on Qwen2.5-3B using a custom C++ runtime with memory-mapped weights, evaluated on CPU without kernel-level optimizations. At 75% compute ratio, we observe 1.15× throughput improvement; output remains coherent on structured completion tasks but exhibits prompt-dependent degradation. At 50% compute ratio, throughput reaches 1.62× but output quality degrades significantly without additional calibration.

We document explicit limitations: quality degradation is non-linear and prompt-dependent; lightweight guardrails (n-gram repetition, entropy monitoring) detect catastrophic failures but not subtle semantic degradation; and the overhead of quality-preserving fallback mechanisms can eliminate net compute savings.

RCC is designed to be architecture-agnostic, though currently validated only on a single model family. This work establishes inference-time compute control as a new system category—orthogonal to compression and acceleration—while acknowledging that practical deployment requires advances beyond this initial formulation.

1 Introduction

Large language model inference presents a fundamental rigidity: once a model is deployed, every forward pass executes the same computational graph regardless of input complexity, output requirements, or resource constraints. A simple completion and a complex reasoning task consume identical compute. This fixed-cost execution model creates inefficiencies across deployment scenarios—from edge devices with strict power budgets to cloud services optimizing cost-per-query.

The research community has developed multiple strategies to reduce inference cost. Quantization and pruning compress model weights, reducing memory footprint and accelerating matrix operations. Knowledge distillation transfers capabilities to smaller architectures. Mixture-of-experts (MoE) activates subsets of parameters per token, but requires architectural changes during training. Speculative decoding and early exit mechanisms attempt dynamic compute allocation, but operate within narrow bounds or require model modifications.

These approaches share a common characteristic: they optimize *what the model is* rather than *how the model executes*. Compression techniques produce a new, smaller artifact. Architectural

changes require retraining. The original model checkpoint—the artifact that practitioners have validated, fine-tuned, and deployed—cannot be directly controlled at inference time.

We identify a gap in the current landscape: **the absence of explicit, post-training control over inference-time computation**. Specifically, we ask: *can a system decide how much computation a model performs per forward pass, without modifying weights, without retraining, and without architectural changes?*

This paper introduces Runtime Compute Control (RCC), a system-level approach that addresses this question. RCC operates as an execution layer between the model checkpoint and the inference runtime. It exposes a single parameter—the compute ratio—that determines what fraction of the model's feed-forward computation is evaluated. The model weights remain unmodified; the checkpoint loaded at 100% compute is identical to the checkpoint loaded at 50% compute. Control occurs entirely at execution time.

RCC does not compete with compression or acceleration techniques. It operates on a different axis: *controllability*. A compressed model is permanently smaller. An RCC-controlled model can execute at different compute levels across requests, within a request, or adaptively based on external signals. This capability has no equivalent in current inference systems.

We validate RCC on Qwen2.5-3B using a custom C++ runtime with memory-mapped weights, evaluated on CPU. Our results establish feasibility, not optimality: at reduced compute ratios, throughput increases but output quality degrades in prompt-dependent ways. We characterize these limitations explicitly, documenting where the current approach succeeds and where it requires further development.

The contribution of this work is not a production-ready system. It is the formulation of a new problem—inference-time compute control—and empirical evidence that the problem is tractable. We position RCC as a foundation for future research in adaptive inference, quality-aware scheduling, and hybrid execution regimes.

2 System Overview

2.1 Design Principles

RCC is designed around three invariants:

1. **Checkpoint preservation**: The model weights are never modified. The same checkpoint file is used regardless of compute ratio. No transformation, re-encoding, or preprocessing is applied to weights.

2. **Execution-time control**: Compute reduction occurs during forward pass evaluation, not before. The decision of how much to compute can change between requests or even between tokens within a single generation.

3. **Transparent interface**: The system exposes a single scalar parameter (compute ratio, ranging from 0.0 to 1.0) that controls the fraction of computation performed. No model-specific configuration or per-layer tuning is required from the user.

These invariants distinguish RCC from adjacent approaches. Pruning modifies weights permanently. Quantization re-encodes weights into lower precision formats. Mixture-of-experts requires architectural changes and routing logic trained into the model. RCC operates on unmodified weights at execution time.

2.2 System Interface

From the user's perspective, RCC interposes between the model checkpoint and the inference call:

```
Input:  model_checkpoint, prompt, compute_ratio  [0.0, 1.0]
Output: generated_tokens, throughput_metrics
```

At `compute_ratio = 1.0`, the system performs standard inference—all computational paths are evaluated. As the ratio decreases, fewer paths are evaluated per forward pass. The mapping from ratio to specific compute reduction is handled internally; the user controls only the ratio.

The interface is stateless with respect to compute ratio: each inference call specifies its own ratio. This enables adaptive policies where different requests, or different phases of generation, execute at different compute levels.

2.3 Scope of Control

RCC currently targets feed-forward network (FFN) computation within transformer blocks. In standard transformer architectures, FFN layers constitute approximately two-thirds of total floating-point operations per forward pass. Attention computation remains unmodified in the current implementation.

This scope is a design choice, not a fundamental limitation. FFN layers exhibit structural regularity (large matrix multiplications with element-wise activations) that admits efficient partial evaluation. Attention mechanisms involve more complex dependencies (key-value interactions across sequence positions) that complicate selective execution. Extending control to attention is a direction for future work.

2.4 What RCC Does Not Do

To prevent misinterpretation, we explicitly state what RCC is not:

- **Not compression**: Model size on disk and in memory is unchanged. A 3B model at 50% compute ratio still loads 3B parameters.
- **Not acceleration**: RCC does not optimize kernel execution or memory access patterns. Speedup comes solely from executing fewer operations, not from executing the same operations faster.
- **Not quality-preserving by construction**: Reducing computation affects model output. RCC provides control, not guarantees. Quality preservation requires external mechanisms (calibration, guardrails, fallback policies) that are outside RCC's core scope.
- **Not a training method**: RCC applies post-training. It does not involve fine-tuning, distillation, or any gradient-based optimization.

2.5 Runtime Architecture

The RCC runtime consists of three components:

1. **Checkpoint loader**: Memory-maps the original model weights without transformation. Supports standard formats; no custom serialization required for the model itself.
2. **Execution controller**: Receives the compute ratio and determines which computational paths to evaluate for each forward pass. The controller maintains internal state for consistency within a generation but exposes no parameters beyond the ratio.
3. **Inference engine**: Executes the transformer forward pass, skipping computation for paths not selected by the controller. The engine is responsible for correct output despite partial evaluation.

The current implementation is a custom C++ runtime optimized for validation, not production throughput. It demonstrates that the architecture is viable; performance engineering is future work.

3 Evaluation

3.1 Experimental Setup

We evaluate RCC on a single model and hardware configuration to establish baseline feasibility. Our goal is not comprehensive benchmarking but demonstration that inference-time compute control produces measurable effects on throughput and output characteristics.

Model: Qwen2.5-3B, a 3-billion parameter transformer with 36 layers, 2048 hidden dimension, and 11008 intermediate dimension in FFN blocks. We use the base (non-instruct) checkpoint without modification.

Runtime: Custom C++ inference engine with memory-mapped weight loading. The implementation prioritizes correctness and validation over throughput optimization. No SIMD vectorization, GPU acceleration, or kernel-level optimizations are applied.

Hardware: Consumer-grade CPU (Intel Core series). All measurements are single-threaded to isolate the effect of compute reduction from parallelization.

Methodology: For each compute ratio, we measure tokens per second (throughput), output coherence on a fixed set of completion prompts, and qualitative assessment of degradation patterns. We do not report perplexity or benchmark scores. These metrics require ground-truth references and measure model quality, not system behavior.

3.2 Throughput Results

Compute Ratio	Tokens/sec	Relative Speedup
100%	0.46	1.00× (baseline)
85%	0.53	1.15×
75%	0.55	1.19×
50%	0.75	1.62×

Table 1: Throughput at varying compute ratios. Measurements on CPU, single-threaded.

Throughput increases monotonically as compute ratio decreases. The relationship is sublinear: reducing compute by 50% yields 1.62× speedup, not 2×. This reflects fixed costs (attention, embedding lookup, sampling) that are unaffected by FFN compute reduction.

Absolute throughput (0.46 tokens/sec at baseline) reflects our unoptimized CPU runtime, not RCC overhead. A production implementation with GPU acceleration and optimized kernels would show higher absolute throughput; the relative speedup from compute reduction would remain comparable.

3.3 Output Characteristics

We evaluate output coherence on 10 completion prompts spanning English and Spanish, technical and narrative styles. Coherence is assessed qualitatively: does the output form grammatically valid, topically relevant continuations?

Key observations: (1) Degradation is non-linear: quality drops sharply between 85% and 75%, not gradually across the range. (2) Degradation is prompt-dependent: some prompts remain coherent at 75% while others degrade at 85%. We found no reliable predictor of which

Compute Ratio	Coherent	Degradation Pattern
100%	10/10	None (baseline)
85%	7/10	Occasional repetition, topic drift
75%	5/10	Frequent repetition loops, reduced vocabulary
50%	2/10	Severe degradation, nonsensical output

Table 2: Output coherence at varying compute ratios across 10 test prompts.

prompts are robust. (3) Failure modes are detectable but not preventable: repetition loops and entropy collapse are observable post-hoc.

3.4 Guardrail Experiments

We implemented lightweight runtime guardrails to detect degradation and trigger fallback to 100% compute: n-gram repetition detection, output entropy monitoring, and token novelty rate.

Results: Guardrails successfully detect catastrophic failures (infinite loops, character-level repetition). However, subtle semantic degradation passes undetected, and fallback overhead can exceed the compute savings from reduced-ratio generation. At 85% compute with guardrails, effective compute consumption was 1.17× baseline due to fallback frequency.

This finding is significant: **quality-preserving fallback mechanisms can eliminate the efficiency gains they are designed to protect**. Guardrails provide safety, not savings.

4 Limitations and Discussion

4.1 Current Limitations

We identify five limitations of the current RCC implementation:

L1. Quality degradation is unpredictable. Output quality does not degrade smoothly with compute ratio. Some prompts tolerate 75% compute; others fail at 90%. We found no reliable signal that predicts robustness.

L2. Guardrails detect symptoms, not causes. Our runtime guardrails identify repetition loops and entropy collapse after they begin. They cannot predict degradation before it occurs, nor detect subtle semantic drift.

L3. Fallback overhead can exceed savings. When guardrails trigger regeneration at 100% compute, the total cost includes both the failed partial generation and the full regeneration. At 85% compute with active guardrails, our measured effective compute was 1.17× baseline.

L4. Single-model validation. All results are from Qwen2.5-3B. We have not validated whether findings generalize to other model families, scales, or architectures.

L5. CPU-only implementation. Our runtime demonstrates feasibility but not production performance. GPU inference introduces different bottlenecks that may change the compute-reduction-to-speedup relationship.

4.2 What RCC Establishes

Despite these limitations, RCC demonstrates:

Feasibility. Post-training, execution-time compute control is technically viable. A system can reduce FFN computation without modifying weights and produce coherent output under favorable conditions.

Measurability. The relationship between compute ratio and system behavior is observable and quantifiable. RCC provides a control surface that admits empirical study.

Separability. Compute control is orthogonal to compression and acceleration. RCC can, in principle, be combined with quantized models or optimized runtimes.

4.3 Future Directions

RCC opens several research questions: adaptive policies for dynamic ratio adjustment, calibration to improve robustness, extension to attention mechanisms, and quality prediction before generation. We leave these to future work.

4.4 Conclusion

Runtime Compute Control addresses a capability gap in LLM inference: the absence of explicit, post-training control over computation. We demonstrate that such control is feasible, describe a system architecture that provides it, and characterize both its effects and its limitations.

RCC is not a production system. It is a formulation of a problem and evidence that the problem is tractable. The value of this work lies not in the speedups achieved—which are modest and fragile—but in establishing inference-time compute control as a valid research direction and system design axis.

We release this work to document priority, invite scrutiny, and encourage others to address the limitations we have identified.