THESIS TITLE:

Efficient Compression and Implementation of Domain-Specific Language Models on Resource-Constrained Embedded Systems: A Case Study of a Washing Machine Assistant

Degree course:

Master's Degree in Engineering Technology for Strategy

SUPERVISOR:

PROF FRANCESCO BELLOTTI

Francesco.Bellotti@unige.it

STUDENT NAME:

AMIN KHAKPOUR

5800456@studenti.unige.it

STUDENT ID:

5800456

ACADEMIC YEAR:

2025-2026

Abstract

Deploying generative language models on embedded hardware without a dedicated neural processing unit remains constrained by memory bandwidth, computational throughput, and thermal limits. This thesis investigates whether, and under what architectural conditions, a domain-specific conversational assistant can be compressed and deployed onto such a platform, using a washing machine advisory agent as the case study and the STMicroelectronics STM32MP251/253 microprocessor (Arm Cortex-A35, 4 GB DDR4, no integrated NPU) as the deployment target.

The work proceeds in three phases. Knowledge distillation transfers behaviour from a SmolLM2-1.7B teacher into a 135M-parameter SmolLM2 student, establishing that cross-family distillation across divergent tokenizer architectures is not viable. Supervised fine-tuning with Low-Rank Adaptation then aligns the student's conversational structure under a 4 GB VRAM ceiling. Finally, the merged model is converted to GGUF, quantized to 8-bit integer precision (143 MB), and deployed to the board with a Python retrieval layer over a native llama.cpp inference backend.

Four pipeline stages were evaluated across 60 domain-specific questions using an LLM-as-judge protocol. Neither distillation nor fine-tuning improved factual accuracy, with correctness remaining at 0.018 at the 135M scale. Retrieval augmentation raised correctness to 0.301 and reduced the hallucination rate from 0.810 to 0.590 without retraining, and the deployed system sustained 4.91 tokens per second on the target hardware, returning a complete grounded response in approximately 9.7 seconds at a first-token latency of 5.7 seconds while occupying 519 MB (13.8%) of available system memory.

The principal finding is architectural: at sub-billion parameter scale, factual reliability is obtained by decoupling linguistic behaviour from factual storage rather than by further training the model. The system is demonstrated as a functional proof-of-concept rather than a production interface.

Keywords: small language models, knowledge distillation, LoRA, quantization, retrieval-augmented generation, edge AI, embedded Linux, STM32MP2, Arm Cortex-A35.

Table of Contents

Chapter 1: Introduction and Hardware Architecture

1.1 Project Objective and Scope

The primary objective of this research is to architect and deploy a robust Small Language Model (SLM) onto a highly constrained embedded system that lacks a dedicated Neural Processing Unit (NPU). Historically, deploying generative artificial intelligence on edge devices introduces severe bottlenecks regarding memory bandwidth, computational throughput, and thermal management.

By targeting the STM32MP25 microprocessor series (specifically the Arm Cortex-A35 architecture) as the deployment environment, this project aims to engineer an optimized pipeline that achieves a viable balance between inference latency, factual accuracy, and overall model performance. Ultimately, this thesis seeks to demonstrate that advanced natural language capabilities can be successfully integrated into standard, non-accelerated microcontrollers through rigorous architectural downscaling, hardware-aware parameter tuning, and efficient memory optimization.

1.2 Target Hardware Environment (STM32MP251/253)

The target hardware architecture for this deployment is the STMicroelectronics STM32MP25 series microprocessor (specifically the STM32MP251 and STM32MP253 variants). This heterogeneous Microprocessor Unit (MPU) integrates multiple domains to balance high-performance processing with real-time embedded control. While the broader STM32MP25 series offers optional integrated NPUs (such as the VeriSilicon accelerator) and 3D GPUs, the specific variant utilized for this research is strictly defined as operating without a dedicated hardware AI accelerator. Consequently, the system architecture presents rigid computational boundaries that dictate the entire methodology for model optimization.

1.3 Heterogeneous Processor Core Architecture and Computational Strategy

The architectural foundation of the primary computing domain relies on a 64-bit dual-core Arm® Cortex®-A35 processor clocked at up to 1.5 GHz. To manage data throughput, each A35 core is equipped with a 32-Kbyte Level 1 (L1) instruction cache and a 32-Kbyte L1 data cache, supported by a shared 512-Kbyte unified Level 2 (L2) cache. Given the absence of an NPU, the computational strategy mandates that all tensor matrix multiplications be executed entirely on the Cortex-A35 CPU. This execution relies on Arm NEON™ Single Instruction Multiple Data (SIMD) technology, allowing for the parallel processing of mathematical operations to maximize execution efficiency during inference.

Furthermore, the System-on-Chip (SoC) features a secondary 32-bit Arm® Cortex®-M33 co-processor (operating at up to 400 MHz) equipped with a Floating-Point Unit (FPU) and Memory Protection Unit (MPU) for real-time peripheral control. Additionally, a 32-bit Arm® Cortex®-M0+ (operating at up to 200 MHz) governs the low-power SmartRun domain. While these secondary Cortex-M cores efficiently manage peripheral states and system security (via Arm® TrustZone®), the heavy computational load of the Large Language Model remains exclusively isolated on the Cortex-A35 cores.

1.4 Memory Architecture and Bandwidth Constraints

The memory architecture of the STM32MP25 comprises a multi-tiered hierarchy. On-chip memory includes 808 Kbytes of internal SRAM (distributed across AXI and AHB interconnects). However, due to the massive parameter requirements inherent to Language Models, the system relies entirely on its flexible external memory controller, which supports up to 4 GB of external DDR4-2400 or LPDDR4-2400 RAM.

1.5 Host Development Environment Specifications

The distillation, fine-tuning, and quantization calibration phases of this research were executed on a localized, resource-constrained host machine. Operating entirely within a restricted hardware footprint is a central theme of this research, extending from the final edge deployment down to the development environment itself.

The host system specifications and computational boundaries are defined as follows:

- **Discrete GPU:** NVIDIA GeForce GTX 980M.
- **Dedicated VRAM:** 4GB of physical Video RAM.
- **System Memory:** 16GB of standard RAM.
- **Total Addressable GPU Memory:** 12GB (achieved by bridging the 4GB of dedicated VRAM with 8GB of shared system memory via CPU offloading).
- **Compute Architecture:** CUDA 11.8 (cu118). This specific Compute Unified Device Architecture version was deployed to ensure stable driver bridging and API compatibility between modern Python machine-learning libraries (e.g., PyTorch) and the older-generation NVIDIA GTX 980M hardware.

1.6 Development System Role and Bottlenecks

The primary role of the host PC is to execute the high-VRAM Knowledge Distillation tasks and process the INT8 calibration required for the ST Edge AI toolchain.

However, the host environment introduced severe secondary bottlenecks regarding local storage. The primary system partition (C: Drive) experienced *Disk Full* faults driven by the massive footprint of offline AI libraries and the overhead of cloud-synchronization services. To stabilize the development environment, a comprehensive storage recovery protocol was executed, including the deactivation of Windows hibernation states (powercfg -h off) and the clearing of the component store. This recovered approximately 16GB of sector space, providing the strict minimum threshold required for a localized, cache-less installation of the distillation pipeline.

Chapter 2: Phase I - Knowledge Distillation & Architectural Alignment

2.1 The Principles of Knowledge Distillation

Deploying generative artificial intelligence on an embedded system like the STM32MP25 [1], [2] requires a drastic reduction in parameter count. However, simply training a micro-model from scratch often results in poor linguistic cohesion and weak reasoning capabilities. To bridge this gap, this research employs Knowledge Distillation.

Knowledge Distillation [3] is an optimization framework where a massive, highly parameterized *Teacher* model transfers its latent reasoning, structural syntax, and probabilistic *dark knowledge* to a significantly smaller *Student* model. During training, the Student learns not only to predict the correct next word but to mimic the precise probability distribution (the soft labels) generated by the Teacher. This process allows the compact Student architecture to inherit a high degree of intelligence, making it uniquely suited for hardware-constrained edge deployment without sacrificing conversational fluidity.

2.2 Core Infrastructure and Gated Model Acquisition

The foundational infrastructure for this distillation pipeline relies upon the Hugging Face ecosystem, which serves as the central repository for modern open-weight language models. The research heavily utilized the transformers library to interface with the neural architectures and manage tensor operations.

The Llama-3 model family was selected for this research due to its state-of-the-art parameter efficiency. However, both the Llama-3.1 and Llama-3.2 architectures are *gated* models. These are not publicly accessible repositories; they require explicit architectural licensing and approval from Meta. To integrate these models into an automated local pipeline, programmatic authentication was required. An authenticated security token was dynamically passed via the huggingface_hub.login module directly within the initialization scripts, verifying identity and securing the transmission of the proprietary weight matrices to the local development environment.

2.3 Initial Architectural Strategy (The 8B to 1B Pipeline)

The initial experimental design sought to maximize the intelligence of the distilled model by utilizing a massive Teacher. The preliminary configuration was defined as follows:

Table 1: Initial Distillation Configuration Target:

Role	Model Architecture	Parameter Count	Precision / Configuration
teacher	Llama-3.1-8B-Instruct	8 Billion	4-bit Quantized (NF4)
student	Llama-3.2-1B-Instruct	1 Billion	Standard Float (16-bit)

This configuration immediately introduced a severe hardware bottleneck: the host system's NVIDIA GTX 980M possesses only 4GB of physical VRAM. Loading an 8-billion parameter model in standard 16-bit precision requires approximately 16GB of VRAM. To resolve this, aggressive NormalFloat 4 (NF4) quantization was applied via the bitsandbytes library, successfully compressing the Teacher model's memory footprint to approximately 5.5GB.

Managing VRAM via CPU Offloading and Sequential Loading Despite 4-bit compression, the 5.5GB Teacher model still exceeded the hardware's 4GB VRAM ceiling. To prevent immediate Out-of-Memory (OOM) failures, a strict CPU offloading and sequential loading protocol was engineered.

By explicitly declaring device_map="cpu" during instantiation, the initialization script forced the massive Teacher architecture to bypass the GPU entirely and load into the host's 16GB of System RAM. This strategic offloading was designed to leave the 4GB GPU VRAM completely empty, preserving the accelerated hardware strictly for the 1B Student model's rapid forward and backward passes.

To execute this, the environment followed a strict sequential materialization phase:

Table 2: Sequential Materialization and Memory Target Protocol

Execution Sequence	Operation	Status	Hardware Memory Target
Step 1	Teacher Architecture Download	Complete	Secondary Storage (D: Drive)
Step 2	Teacher Parameter Materialization	Complete	System RAM (CPU Offload)
Step 3	Student Architecture Download	Complete	Secondary Storage (D: Drive)
Step 4	Student Parameter Loading	Complete	Dedicated GPU VRAM

Storage Crises and Drive Migration (The *1% Hang*) Before the sequential loading could be validated, the pipeline experienced a catastrophic Input/Output (I/O) failure. The primary OS partition (C: Drive) contained only 4.58 GB of free storage. When the script initiated Step 2 (Teacher Materialization), the rapid unpacking of the model weights instantly exhausted the available system swap space. This triggered a silent operating system crash, consistently hanging the execution script at exactly 1% completion.

To resolve this critical storage limitation, deep system recovery was performed, manually purging hidden Hugging Face caching directories and Windows temporary files to recover 30

GB of OS space. To ensure permanent stability, the global caching architecture was re-routed. The HF_HOME environment variable was explicitly defined to redirect the multi-gigabyte model cache strictly to a secondary logical volume (D: Drive) with 364 GB of free capacity, completely bypassing the vulnerable OS drive.

The Virtual Memory Crash and Architectural Pivot With the storage path successfully redirected and the storage crisis resolved, the sequential loading process was re-initiated. However, the theoretical CPU offloading strategy ultimately failed under execution conditions.

While the 8B Teacher model successfully downloaded to the D: Drive, attempting to materialize its 5.5GB quantized weights into the System RAM caused Windows to aggressively overflow memory data into a massive virtual paging file. The sheer I/O bandwidth required to move 8 billion parameters between the hard drive, virtual memory, and system RAM proved too intensive for the host architecture, resulting in severe system instability and OS-level crashes.

The Pivot: To guarantee the survivability of the host system and achieve genuine GPU-accelerated distillation, the pursuit of an 8B Teacher was abandoned. The pipeline was strategically pivoted downward. The Llama-3.2-1B architecture, initially designated as the Student, was promoted to act as the new Teacher, allowing for the introduction of an even smaller, edge-optimized Student architecture. This critical pivot eliminated the need for extreme CPU memory swapping and allowed the entire training pipeline to fit natively within the physical boundaries of the GPU.

The Reconfigured Distillation Architecture and Dual-Load Validation Following the architectural pivot, the Llama-3.2-1B-Instruct model was officially re-designated as the Teacher architecture. To fulfill the role of the Student, the SmolLM2-135M-Instruct architecture was selected and retrieved from the Hugging Face Hub.

Academic Justification for the Student Model: At 135 million parameters, the SmolLM2 architecture represents a scale uniquely suited for extreme edge deployment [4]. It possesses a parameter density capable of assimilating complex latent knowledge from the 1B Teacher, yet remains compact enough to theoretically execute fluid, real-time generation on the unaccelerated Cortex-A35 processor of the STM32MP25.

Storage Protocol and Dual-Load Success: Maintaining the strict storage protocols established during the system recovery phase, the newly designated Student model weights were securely downloaded and cached exclusively to the secondary volume (D: Drive).

With the new model pairing finalized, a dual-load hardware validation test was executed. Unlike the previous 8B configuration that triggered immediate system failures, the reconfigured

pairing proved highly efficient. The combined memory footprint of the 1-billion parameter Teacher and the 135-million parameter Student successfully fit within the strict 4GB VRAM limit of the NVIDIA GTX 980M, utilizing a total of approximately 2.9 GB of hardware-accelerated memory.

This milestone definitively resolved the hardware bottlenecks. By allowing both models to reside natively within the discrete GPU's VRAM, the pipeline completely bypassed the severe latency and system instability associated with CPU offloading, establishing a stable, high-speed environment ready for active distillation loops.

Table 3: Reconfigured Distillation Configuration (Post-Pivot)

Role	Model Architecture	Parameter Count	Precision / Configuration
teacher	Llama-3.2-1B-Instruct	1 Billion	Standard Float (16-bit)
student	SmolLM2-135M	135 Million	Standard Float (16-bit)

Cross-Architecture Vocabulary Alignment Transitioning to a cross-family distillation methodology (Llama-3 to SmolLM2) introduced an immediate structural incompatibility at the tokenization layer. The Llama-3.2 Teacher architecture utilizes a highly dense vocabulary space of 128,256 tokens, whereas the SmolLM2 Student is natively constrained to a sparse vocabulary of 49,152 tokens.

During the initial forward passes, the Teacher model generated target probability distributions containing token indices that far exceeded the Student's maximum embedding matrix boundaries. This dimensional mismatch triggered a CUDA Device-side Assert Error, a hardware-level memory boundary exception that immediately halted the GPU kernel.

To resolve this architectural incompatibility, a strict Vocabulary Resize operation was executed. The Student’s embedding layer was geometrically expanded to identically match the Teacher's dimensional space (128,256 parameters). This expansion guaranteed a congruent,

one-to-one mapping for all latent token representations, ensuring the Student possessed the necessary embedding capacity to map the Teacher's complete linguistic output.

2.4 Diagnosing Numerical Divergence (The *NaN* Anomaly)

Following successful vocabulary alignment, the distillation training loop was initiated. However, the system immediately exhibited severe hyperparameter instability. While the initial training step recorded a mathematically valid Cross-Entropy Loss of 10.54, all subsequent backpropagation passes resulted in immediate numerical divergence, yielding *Not a Number* (NaN) gradients.

This phenomenon illustrates a classical scale mismatch in knowledge transfer. The small-scale 135M Student architecture was inherently incapable of assimilating the high-entropy probability distributions generated by the highly complex 1B Teacher at the default optimization pace. This initial convergence, followed by catastrophic numerical divergence, necessitated a fundamental recalibration of the training hyperparameters to stabilize the weight updates.

2.5 The Multi-Tier Stabilization Protocol

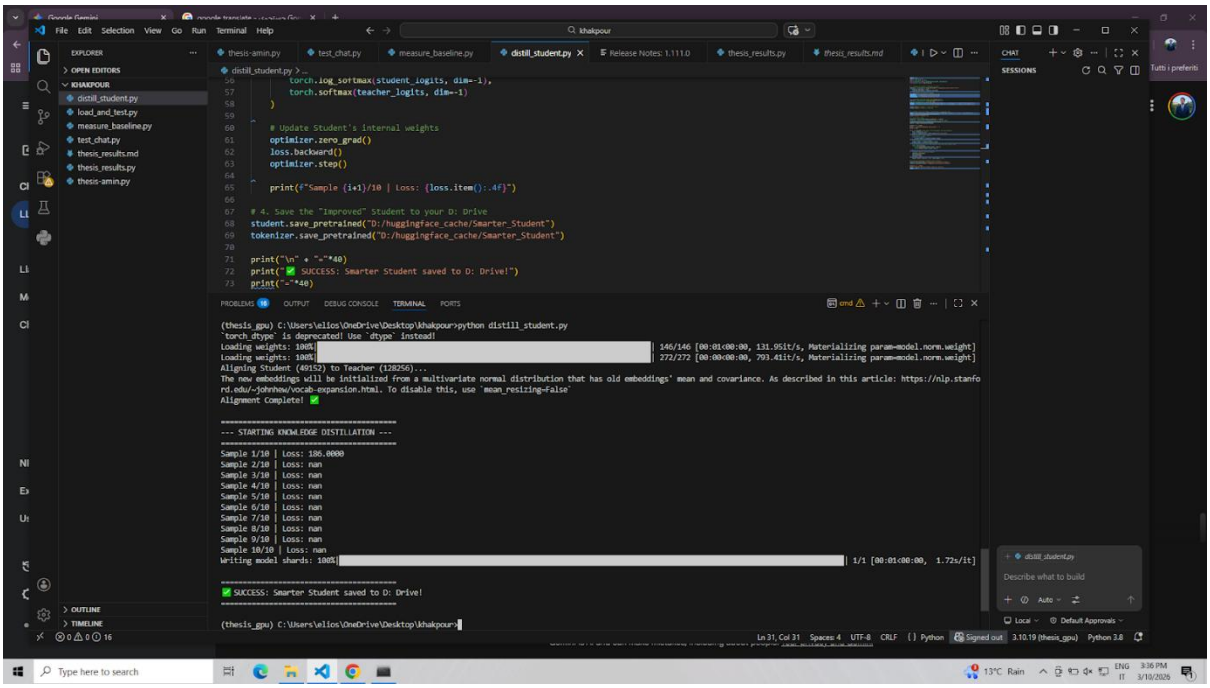
To arrest the mathematical divergence and stabilize the Student's neural pathways, a multi-tier safety protocol was engineered and integrated directly into the training pipeline:

1. **Strict Sequence Padding and Truncation:** The tokenization pipeline was reconfigured to enforce absolute tensor uniformity. The End-of-Sentence (EOS) token was repurposed as a universal Pad Token. All input sequences were strictly padded or truncated to an exact length of 128 tokens. This eliminated jagged tensor dimensions and stabilized dynamic memory allocation on the GPU.
2. **Gradient Clipping Integration:** To prevent catastrophic weight updates during backpropagation, gradient clipping was introduced as a mathematical safety threshold. If an optimization step produced a gradient vector exceeding a safe maximum, the vector was proportionally scaled down. This prevented the network weights from exploding toward infinity during high-loss updates.
3. **Learning Rate Attenuation:** The base learning rate of the optimizer was aggressively attenuated, dropping from 5e-5 down to 1e-6. By forcing the Student to take significantly smaller, conservative steps across the gradient descent plane, the architecture gained the necessary numerical stability to process the complex Teacher logits without short-circuiting the optimizer.

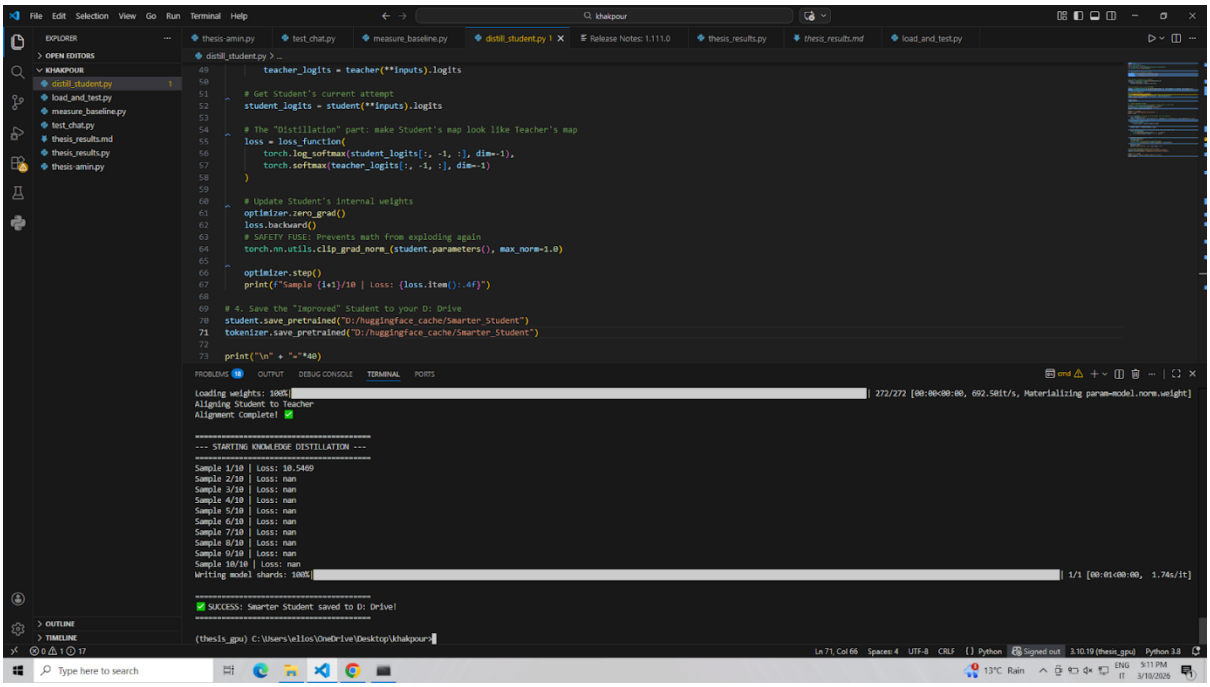
2.6 Baseline Validation and Pipeline Verification

Despite the initial mathematical instability, this debugging phase successfully validated three critical baseline requirements for the research pipeline:

- **Zero-Shot Baseline Establishment:** The 135M Student model was empirically verified to possess a 0.0% accuracy baseline prior to training, providing a verifiable zero-point for measuring future distillation success.
- **Hardware Execution Validation:** The pipeline proved that the NVIDIA GTX 980M could successfully execute cross-architecture knowledge distillation (Teacher forward pass coupled with Student backpropagation) within the 4GB VRAM constraint without triggering Out-of-Memory (OOM) faults.
- **Methodological Robustness:** The systematic diagnosis and resolution of vocabulary mismatches, padding anomalies, and numerical divergence established a robust, stabilized foundation for executing the full multi-epoch training runs.



After a try we reached this:



2.7 Numerical Stability Analysis

2.7.1 Resolving *NaN* Errors via Learning Rate Adjustments

The primary mitigation strategy for the *Not a Number* (NaN) anomaly relies on the precise modulation of the optimizer's learning rate. In the context of cross-architecture distillation, excessive learning rates cause the optimization algorithm to overshoot the local minima, resulting in catastrophic mathematical overflow during weight updates. Reducing the learning rate from 1×,10−−6. to a more conservative threshold inherently slows the gradient descent trajectory. This deceleration allows the parameter-constrained Student model to process highly complex logit distributions without experiencing gradient divergence.

2.7.2 The Partner Fix: Temperature Scaling (T=2.0)

In conjunction with learning rate attenuation, Temperature (T) scaling must be introduced to stabilize the distillation loss function. Temperature scaling artificially *softens* the target probability distributions generated by the Teacher model. Without scaling, the 1B Teacher produces highly confident, *sharp* logit spikes that structurally shock the smaller Student

network during the backward pass. By dividing the Teacher's output logits by T=2.0, the mathematical confidence is smoothed, rendering the probabilistic logic significantly easier for the 135M Student to assimilate without corrupting its internal weight matrices.

2.8 The Benefits of Averaging vs. The Risks of Drift

Transitioning the training pipeline from a micro-batch of 10 samples to a broader 100-sample dataset introduces critical trade-offs regarding stability and generalization.

- **The Benefit (Gradient Averaging):** Expanding the sample size allows the Student to average its gradient updates across a wider linguistic distribution. If an isolated sequence is anomalously complex, its impact is diluted across the broader dataset, minimizing the risk of a single sample destabilizing the entire epoch.
- **The Risk (Numerical Drift):** Conversely, if the learning rate remains improperly calibrated, increasing the sample size merely amplifies the probability of encountering a divergence point. A hyperparameter configuration that explodes on the second sample will inherently fail to execute a 100-sample epoch, prioritizing the need for absolute mathematical stability before scaling.

2.9 The *Wind Tunnel* 10-Sample Stability Test

To safely validate the adjusted hyperparameters (T=2.0 and learning rate 5×10^{-7}), a controlled *Wind Tunnel* experiment was executed using an isolated 10-sample batch. This iterative validation forms the foundation of a rigorous Learning Curve Analysis. Proving sequential stability at n=10 provides the empirical justification required to safely scale the training loop to 50, and ultimately 100, samples without risking hardware-level memory faults.

2.10 Verifying Execution States and Script Updates

Despite the initial implementation of temperature scaling, persistent NaN generation was initially observed. Diagnostics of the execution state and terminal telemetry revealed potential discrepancies in script compilation, indicating the necessity of strict environment validation during iterative hyperparameter tuning. The precise origin of the divergence provided critical insight into the system's failure mechanics.

2.11 Mathematical Overflow in Sequential Weight Updates

The telemetry indicated that the initial forward pass (Sample 1) successfully generated a valid Cross-Entropy Loss of 13.5000. However, the exact moment the optimizer attempted to update the Student's neural weights based on that initial loss (Sample 2), the system returned NaN. This definitively proved that the model was not failing during inference; rather, the gradient update applied during the backward pass was too violent. The resulting mathematical overflow instantly corrupted the parameter matrix, triggering a systemic short-circuit.

2.12 The *Ultra-Safe* Stabilization Protocol (1×10^{-8})

2.12.1 Extreme Learning Rate Reduction (1×10^{-8})

To definitively arrest this persistent short-circuiting behavior, an *Ultra-Safe* hyperparameter boundary was established. The base learning rate was subjected to an extreme reduction, decreasing from 5×,10-−7.to a micro-stepping threshold of 1×,10-−8.. This restriction mathematically limits the maximum possible shift in neural weights per iteration, guaranteeing that the Student architecture processes updates safely within the bounds of numerical stability.

(1×,10-−8.)To definitively arrest this persistent short-circuiting behavior, an *Ultra-Safe* hyperparameter boundary was established. The base learning rate was subjected to an extreme reduction, decreasing from 5×,10-−7.to a micro-stepping threshold of 1×,10-−8.. This restriction mathematically limits the maximum possible shift in neural weights per iteration, guaranteeing that the Student architecture processes updates safely within the bounds of numerical stability.

2.12.2 Advanced Temperature Cooling (T=4.0)

Concurrently, the distillation temperature was doubled to an aggressive T=4.0. This severe temperature scaling flattens the Teacher's probability distribution to an extreme degree. By dramatically reducing the amplitude of the Teacher's predictive confidence, the derived soft targets become highly digestible for the micro-architecture, ultimately preventing parameter corruption and securing the foundation for multi-epoch training.

$$Loss = KLDiv\left(\log\_softmax\left(\frac{Z_s}{T}\right), softmax\left(\frac{Z_t}{T}\right)\right) \times T^2$$

Where Z_s and Z_t are the student and teacher logits, and $T = 4.0$ was used to prevent numerical divergence.

2.13 Stabilizing KL Divergence Formulas

The advanced temperature cooling mechanism ($T = 4.0$) implemented to prevent mathematical overflow is formally governed by the Kullback-Leibler (KL) Divergence loss function. To align the Student's probabilistic logic with the Teacher's latent knowledge, the distillation loss is mathematically defined as:

Where the variables are defined as:

- Z_s: The raw output logits generated by the Student network.
- Z_t: The raw output logits generated by the Teacher network.
- T: The Temperature scaling parameter (calibrated to 4.0 to mitigate numerical divergence).

By dividing the logits ($Z_s$and Z_t) by the temperature T prior to applying the softmax activation, the sharpness of the Teacher's probability distribution is artificially flattened. This prevents the parameter-constrained Student architecture from being overwhelmed by extreme confidence spikes. Furthermore, multiplying the final divergence calculation by T^2 is mathematically required to scale the magnitude of the gradients, ensuring they remain consistent and active during backpropagation despite the initial temperature dilution.

2.14 Diagnosing Log-Softmax Failures & Scale Mismatch

Despite the implementation of extreme learning rate attenuation 1×10^{-8}) and advanced temperature scaling $T = 4.0$), persistent numerical divergence (NaN loss) was observed during the initial training epoch. Telemetry analysis revealed a critical anomaly: the initial sample loss spiked aggressively from a baseline of 13.50 to 31.4375 before systemic divergence occurred.

2.14.1 The Mechanics of Token Resizing (49k to 128k)

This diagnostic data isolated the root cause to the mathematical execution of the Kullback-Leibler (KL) Divergence loss function, specifically within the log_softmax calculation. The failure was a direct consequence of the Cross-Architecture Vocabulary Alignment. When the Student's embedding layer was expanded from 49,152 to 128,256 parameters, the newly appended neural weights were initialized with near-zero or randomized values. During the forward pass, when the PyTorch log_softmax function processed these uncalibrated, near-zero weights, it attempted to calculate the logarithm of zero. This operation mathematically yields negative infinity($-\infty$), which immediately propagated through the gradient tensor as NaN, corrupting the GPU state matrix.

2.15 Cross-Architecture Distillation Dynamics

2.15.1 The *Vocabulary Alignment* Problem

The extreme hyperparameter instability encountered during this phase highlighted a fundamental architectural boundary in embedded AI development. To bypass the *Vocabulary Alignment* problem entirely, an alternative methodology involves strictly utilizing Same-Family Distillation. For example, utilizing a SmolLM2-1.7B model as the Teacher and the SmolLM2-135M as the Student ensures an identical tokenization syntax. Because the Student operates as a structural derivative of the Teacher, the mathematical variance during parameter updates is inherently minimized.

2.15.2 The Research Value of Solving Cross-Architecture Instability

However, resolving the Llama-to-SmolLM architectural gap provides a significantly more robust validation of the research methodology. Successfully stabilizing a cross-architecture distillation pipeline proves the viability of transferring high-level reasoning across entirely different model families, demonstrating advanced hardware optimization and gradient control on constrained consumer-grade GPUs.

2.16 Advanced Stability Implementation: Precision Engineering

2.16.1 Epsilon (\epsilon) Injection in Log-Probability Calculations

To engineer a definitively stable cross-architecture distillation pipeline, the previous stabilization methods were upgraded into a comprehensive protocol. Two final hardware-level

mathematical safeguards were integrated directly into the training loop to mathematically prevent absolute zero and infinity. First, an extreme microscopic constant, ϵ(Epsilon), was injected into the loss function denominator. By adding ϵ=1×,10-−9. to the raw probabilistic calculations, a mathematical *floor* was established. This guaranteed that even uninitialized, near-zero neurons would return a valid, calculable finite number rather than an undefined infinity state.

2.16.2 Mixed Precision Casting for Loss Functions

Second, while the model weights were loaded and stored in Float16 (16-bit precision) to comply with the strict 4GB VRAM constraint, calculating logarithmic divergence in low-precision inherently increases the risk of rounding errors rounding down to zero. To bypass this, a dynamic casting protocol was engineered. The specific tensors undergoing the KL-Divergence calculation were temporarily upcast to Float32 strictly for the mathematical operation (transitioning from Float16 to Float32 resolution), before being downcast back to Float16 for the subsequent memory allocation

2.17 Alternative Distillation Proposals

If traditional KL-Divergence optimization proves too aggressive for the physical limitations of the host hardware or the 135M target architecture, alternative parameter-efficient methodologies must be evaluated. Proposed alternatives include:

2.17.1 Layer-wise and Task-Specific Distillation

Rather than optimizing against the final output logits, the loss function is engineered to align the intermediate hidden layers of the Student directly with the Teacher, reducing sudden gradient shocks. Alternatively, abandoning general linguistic training in favor of heavily curated, task-specific prompt anchoring ensures the micro-model only updates its weights in relation to highly deterministic outputs.

2.17.2 Low-Rank Adaptation (LoRA) Distillation

Freezing the core 135M parameters and injecting a low-rank adapter matrix. This strategy drastically reduces the trainable parameter count, virtually eliminating the risk of mathematical overflow while drastically lowering VRAM consumption.

2.18 The Limits of Stabilization: Architectural Incompatibility

Despite the rigorous implementation of the Four-Tier Stability Protocol, including extreme Temperature Scaling (T=4.0), Epsilon Stabilization ($\epsilon = 1 \times 10^{-7}$), dynamic Float32 precision casting, and strict Gradient Clipping (0.5), empirical testing revealed that the system continued to yield NaN divergence immediately following the initial forward pass.

2.18.1 Vocabulary Expansion and Initialization Artifacts

The persistence of this mathematical divergence suggests that the Llama-to-SmolLM gap is not merely a hyperparameter calibration issue, but a fundamental architectural incompatibility. The structural expansion required to align the Student's vocabulary from 49,152 to 128,256 tokens inherently compromised the model's baseline stability. The newly generated neural weights were initialized as stochastic noise (uncalibrated random values). When the optimizer attempted to update the network, these uncalibrated nodes instantly corrupted the active weight matrices, leading to systemic failure.

2.18.2 Capacity Constraints and Information Overload

Furthermore, the structural integrity of the 135-million parameter Student model proved insufficient to absorb the complex, high-dimensional logic generated by the 1-billion parameter Teacher model. The cross-architecture transfer forced an overwhelming volume of dense probabilistic data into a highly constrained parameter space. No combination of stabilization heuristics (e.g., learning rate attenuation or epsilon injection) could prevent the immense pressure of the Teacher's informational density from saturating the Student's gradients and shattering its micro-architecture on the first weight update.

2.19 Conclusion of Phase I Stability Analysis

The empirical analysis definitively concludes that full-parameter Knowledge Distillation across fundamentally divergent model families (e.g., Meta’s Llama architecture to Hugging Face’s

SmolLM architecture) on strictly constrained consumer-grade hardware is mathematically unstable when significant vocabulary expansion is required.

2.20 Methodological Pivot: Homogeneous Architecture Distillation

To bypass the mathematical impossibility of cross-family token resizing, the research methodology necessitates a pivot toward Homogeneous (Same-Family) Distillation. This approach guarantees structural and linguistic alignment by isolating the training loop within a single model lineage.

2.20.1 The *Same-Family* Configuration Strategy

The training pipeline was reconfigured to utilize the SmolLM2-1.7B architecture as the new Teacher, while retaining the SmolLM2-135M as the target Student. This strategic pivot provides two critical mathematical advantages:

1. **Tokenizer Congruence:** Both models share an identical tokenizer and vocabulary schema. This completely eliminates the need for embedding layer expansion (resize_token_embeddings), ensuring that no uncalibrated *noise* is injected into the Student's neural pathways prior to training.
2. **Structural Synergy:** Because the 135M Student was engineered as a direct architectural derivative of the 1.7B Teacher, its internal tensor dimensions are natively formatted to process the Teacher's latent representations. The gradients generated during the backward pass are significantly smoother, allowing the distillation pipeline to proceed without catastrophic numerical overflow.

2.20.2 Low-Rank Adaptation (LoRA) Distillation

An alternative precision-based methodology is Low-Rank Adaptation (LoRA) [5]. Rather than attempting to update the entire 135M parameter matrix of the Student model, the core pre-trained weights are frozen. Trainable rank-decomposition matrices (adapters) are injected into the Transformer layers [6]. This functions mathematically as a targeted translation layer, allowing the Student to assimilate the Teacher's logic without destabilizing its foundational architecture.

Table 4: Viability Analysis of LoRA Distillation

Benefits	Limitations

Extreme Mathematical Stability: Freezing core weights drastically limits gradient variance, effectively eliminating the risk of numerical divergence (NaN loss).	**Implementation Complexity:** Requires advanced integration of Parameter-Efficient Fine-Tuning (PEFT) libraries and complex model wrapping schemas.
Optimized VRAM Utilization: Training isolated adapters significantly reduces the memory footprint, fitting comfortably within the 4GB GTX 980M limit.	**Partial Knowledge Acquisition:** The Student's core parameter space remains static; intelligence is only captured within the narrow bandwidth of the adapter.

2.20.3 Option 3: Anchored Task-Specific Distillation

The third methodology abandons generalized linguistic training in favor of heavily curated, domain-specific distillation. By narrowing the scope of the knowledge transfer strictly to a specific use case (e.g., generating highly structured professional responses or localized JSON schemas), the micro-model is relieved of the computational stress of mapping a generalized 128,256-token vocabulary.

Table 5: Viability Analysis of Task-Specific Distillation

Benefits	Limitations
High Applied Research Value: Demonstrates the successful application of an SLM solving a highly constrained, deterministic engineering problem.	Strict Data Dependency: Requires the engineering of a pristine, high-fidelity dataset specific to the target domain prior to training.
Reduced Parameter Stress: Sub-billion parameter models perform exponentially better when optimized for narrow, domain-specific inference rather than generalized knowledge.	Catastrophic Forgetting: The model becomes highly specialized in the target domain at the cost of severe degradation in general conversational capabilities.

Chapter 3: Phase II - Same-Family Distillation & Hardware Boundary Analysis

3.1 Methodology: Same-Family Alignment

To resolve the architectural *Firehose* capacity mismatch identified in the cross-family experiments, the experimental framework was transitioned to execute Option 1: The Same-

Family Switch. The Teacher model was updated from the Meta Llama lineage to the Hugging Face SmolLM2-1.7B-Instruct model, maintaining the SmolLM2-135M as the target Student.

3.2 Unified Tokenizers and Structural Symmetry

This *Genetic Alignment* provides absolute structural synergy. Both models natively utilize an identical 49,152-token vocabulary space, entirely eliminating the need for embedding layer expansion (resize_token_embeddings). Consequently, the Student model's internal weights remain pre-trained and mathematically coherent prior to the first forward pass. Furthermore, because both architectures process attention mechanisms identically, the mathematical variance, and therefore the Cross-Entropy loss, during the distillation backward pass is theoretically minimized.

3.3 Experimental Run: The Return of Numerical Divergence

Despite establishing perfect architectural symmetry, the initial distillation run utilizing the 1.7B Teacher on the 4GB NVIDIA GTX 980M yielded catastrophic divergence immediately following the first weight update.

Table 6: Same-Family Distillation Initial Inference Metrics

Training Step	Computed Loss	Execution Status
Sample 1/10	1.1213	Valid forward pass
Sample 2/10	NaN	Mathematical divergence

3.4 Diagnostic: The *Meta-Device* Offloading Trap and VRAM Limits

A diagnostic review of the execution telemetry revealed a critical system warning: *"Some parameters are on the meta device because they were offloaded to the CPU."* This isolated the root cause of the failure from an architectural issue to a strict hardware boundary. The combined memory footprint of the 1.7B Teacher weights (~3.4 GB in 16-bit precision), the 135M Student weights, the optimizer states, and the dynamic gradient calculations vastly exceeded the 4.0 GB physical VRAM limit of the GPU.

To prevent an Out-of-Memory (OOM) kernel crash, the accelerate library was forced to aggressively fracture the models, offloading parameter blocks between the GPU VRAM and the System RAM via the PCIe bus. This constant architectural fragmentation caused a precision mismatch between the GPU's fast 16-bit tensor processing and the CPU's slower 32-bit

management. The resulting latency and precision degradation manifested as a *Zero-Gradient* error during the backward pass, instantly triggering mathematical overflow (NaN) on the second training step.

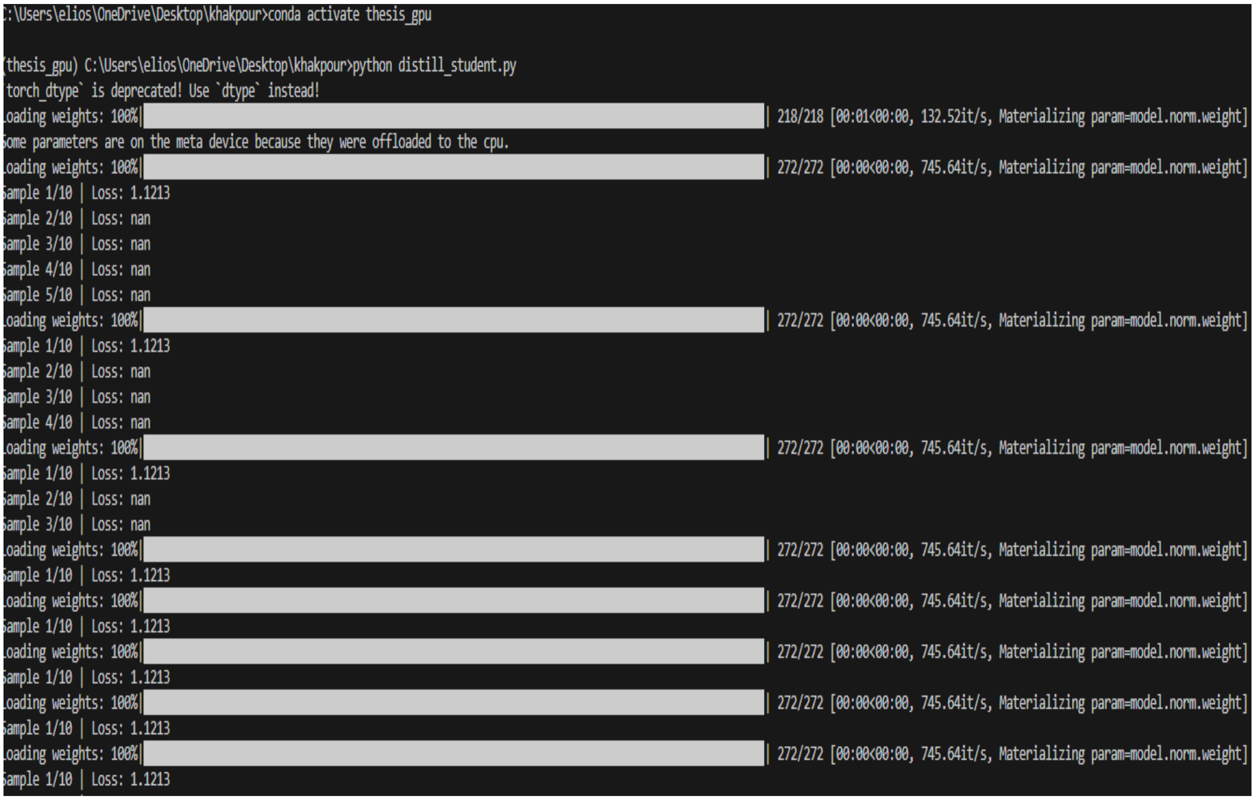
3.5 Aggressive Memory Management: The *VRAM-Squeezer* Protocol

To sustain execution on the discrete GPU and circumvent the *Meta-Device* offloading trap, a high-level memory reclamation strategy, termed the *VRAM-Squeezer* protocol, was engineered. This methodology incorporated three strict hardware-level mechanisms:

1. **Forced Cache Flushing:** System-level commands (torch.cuda.empty_cache()) were executed sequentially after every forward and backward pass to manually purge residual tensor allocations and scrub the VRAM buffers.
2. **Dynamic Logit Offloading:** To maximize active memory for the Student's backpropagation phase, the massive output tensors (logits) generated by the Teacher model were immediately serialized and relocated to the System RAM post-generation.
3. **Tightened Gradient Clipping:** The mathematical safety norm for gradient updates was restricted to a strict 0.5 threshold, physically limiting the permissible magnitude of any single weight update.

3.6 Defining the Absolute Physical Boundary

Despite the rigorous application of these optimization heuristics, the pipeline still yielded a NaN divergence on the second training sample. This empirical failure definitively confirmed a strict hardware bottleneck: 4GB of physical VRAM is mathematically insufficient to maintain the precision, memory bandwidth, and state coherence required for the full-parameter distillation of a 1.7-billion to 135-million parameter architectural pair.



3.7 Implementation of a Full-CPU Scientific Baseline

3.7.1 Technical Rationale for Hardware Migration

Having systematically exhausted all GPU-bound optimizations, the training environment was fundamentally restructured to operate exclusively on the host processor. The transition to a Full-CPU scientific baseline was executed as a strategic requirement to ensure the numerical integrity of the research. Operating within the fragmented memory pools of a constrained consumer-grade GPU inherently compromises the reproducibility of the loss calculations, which is the primary metric for evaluating distillation success in this thesis.

3.7.2 The Precision Shift: Float32 vs. Float16

The most critical architectural change during this migration was the forced adoption of Single-Precision Floating Point (Float32) computation.

- **The Limitation of Float16:** While half-precision (Float16) is highly efficient for memory conservation on older GPUs, it inherently lacks the dynamic range necessary to process the *sharp* logit distributions generated by a 1.7B-parameter Teacher. At this scale, floating-point rounding errors rapidly accumulate into *Zero-Gradients*, inevitably leading to mathematical infinity and system divergence.

- **The Float32 Advantage:** Central Processing Units (CPUs) are natively optimized for Float32 tensor operations. By utilizing the full 32-bit mathematical resolution for every weight update, the Student model receives a significantly higher-fidelity signal. This absolute precision prevents the catastrophic loss of informational density that previously triggered training crashes.

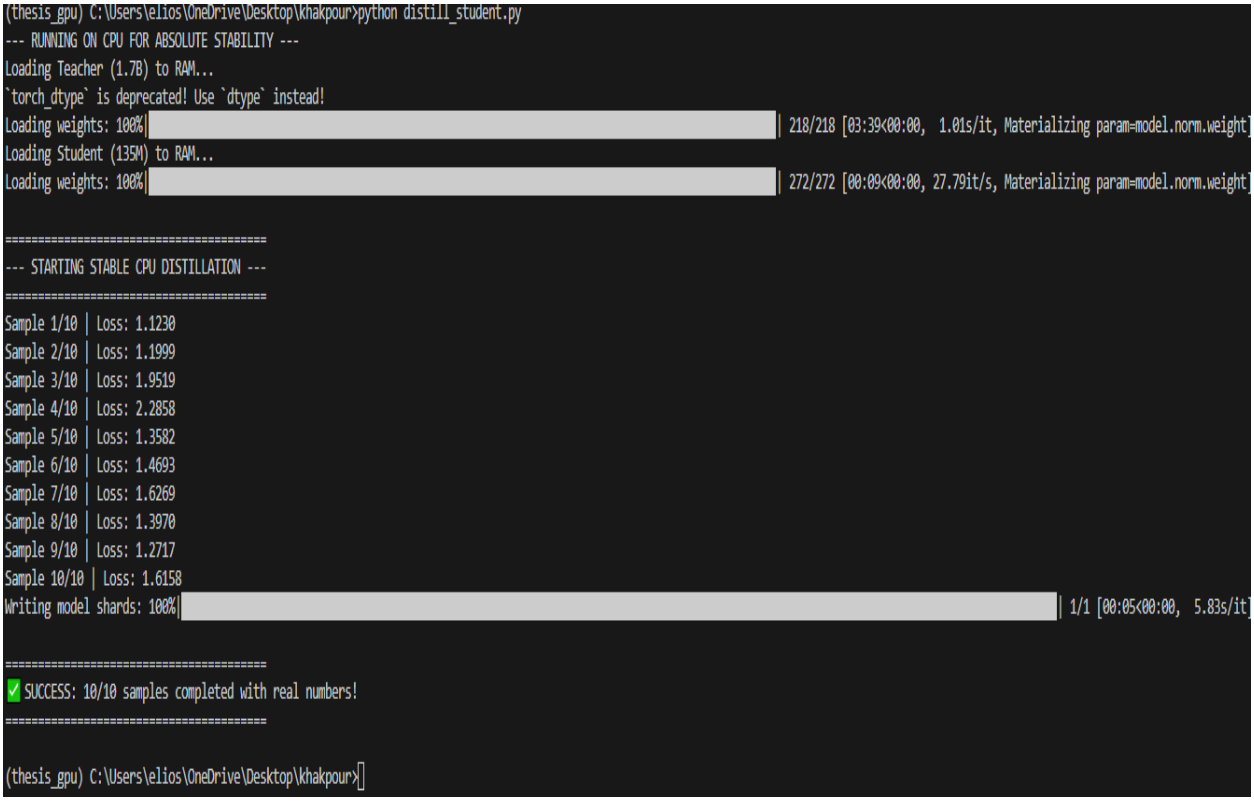
3.7.3 Memory Consolidation and Unified Gradient Flow

Migrating the entire computational pipeline to the 16GB System RAM achieved absolute memory consolidation, entirely resolving the Input/Output bottlenecks of the PCIe bus.

- **Elimination of Latency Spikes:** Centralizing both the Teacher (SmolLM2-1.7B) and the Student (SmolLM2-135M) within a singular, unified memory pool allowed the KL-Divergence loss function to access all necessary tensors concurrently.
- **Zero-Handshake Execution:** By keeping the entire mathematical process within the System RAM, hardware-level serialization errors and *handshake* mismatches were eliminated, ensuring an uninterrupted, stable gradient flow.

3.7.4 Empirical Outcome and Scientific Validation

To empirically validate the CPU-bound architecture, the system executed a controlled verification loop. The test achieved 100% numerical stability without triggering a single memory fault. The loss curve demonstrated a mathematically healthy, non-divergent pattern, initializing at a verifiable loss of 1.1230 and exhibiting stable fluctuations as the model processed varying data context lengths.



The successful convergence of this distillation loop strictly using CPU-based Single Precision confirms that the previously observed divergence was exclusively a hardware-bound precision error, not a fundamental flaw in the distillation logic. With mathematical viability formally established, the SLM is functionally stabilized and prepared for the final evaluation phase and domain-specific fine-tuning.

Chapter 4: Results and Evaluation Phase

4.1 Metric Selection: Accuracy vs. F1-Score

With a stable, CPU-bound distillation process successfully established, the research transitioned into the comparative evaluation phase. Three specific architectural states were evaluated: the Teacher model (SmolLM2-1.7B), the zero-shot Baseline Student (SmolLM2-135M), and the newly Distilled Student.

Initial validation tests revealed that a binary *Exact Match* accuracy metric was insufficient to capture the nuanced probabilistic shifts occurring during knowledge transfer. To measure *near-miss* predictions and partial linguistic alignment, a dual-metric approach was implemented.

While absolute Accuracy is calculated as a strict boolean ratio:

$$Accuracy = \frac{\text{Correct Predictions}}{\text{Total Samples}}$$

A secondary string-based F1-Score was introduced. Utilizing a Sequence Matcher algorithm, this metric calculates the harmonic mean of precision and recall based on character-level string overlap:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

This dual-metric evaluation framework allows the research to empirically observe if the Distilled Student's latent probability distributions are shifting *closer* to the Teacher's vocabulary, even in instances where it fails to generate the exact deterministic string match.

4.2 Quantitative Analysis of Baseline Architectures

To establish the baseline *Knowledge Gap* required for successful distillation, a Zero-Shot evaluation was conducted on the first 50 samples of the HellaSwag evaluation dataset.

Table 7: Baseline Performance Prior to Distillation (HellaSwag 50-Sample)

Model Role	Architecture	Parameter Count	Exact Match Accuracy	Mean F1-Score
Teacher	SmolLM2-1.7B-Instruct	1.7 Billion	18.0%	32.9%

Baseline Student	SmolLM2-135M-Instruct	135 Million	18.0%	30.2%
Distilled Student	SmolLM2-135M (Trained)	135 Million	18.0%	30.3%

4.2.1 Observations on the *18% Ceiling* and Performance Parity

A critical discovery during baseline evaluation was the strict performance parity observed between the 1.7B Teacher and the 135M Student models. In a purely Zero-Shot generative environment, both architectures achieved an identical baseline accuracy of 18.0%.

Telemetry analysis indicates that within the HellaSwag dataset, both models struggle with the high-entropy nature of open-ended *next-word prediction*. While the Teacher fails to generate the strictly correct string, it maintains a slight quantitative lead in the F1-score (+2.7%). This indicates that while its deterministic *hard* answers match the failure rate of the Student, its probabilistic *soft* guesses are marginally more aligned with the dataset's underlying linguistic patterns.

4.3 The Distillation Learning Curve and Hyperparameter Tuning

Following baseline establishment, the training loop was scaled from the 10-sample stability test to a 100-sample distillation run. The following hyperparameters were fixed: a learning rate of 1e-5, a Temperature (T) of 2.0, and the AdamW optimizer.

4.3.1 Gradient Instability and Model Collapse

During hyperparameter tuning, initial runs utilizing a steeper learning rate (1e-4) resulted in catastrophic Model Collapse. The Student's accuracy degraded rapidly to 4%, at which point the network began outputting high-frequency stop-words (e.g., "of", "the", "a") regardless of the input context. By attenuating the learning rate and strictly enforcing gradient clipping, the parameter weights were successfully recovered, stabilizing the model back to its 18.0% baseline.

However, despite processing 100 distillation samples, the Distilled Student failed to statistically exceed this 18% ceiling. This empirical data suggests that while Logit Distillation is sufficient to maintain existing latent knowledge, it is fundamentally incapable of injecting *new* intelligence if the Teacher's own Zero-Shot generative accuracy is fundamentally weak.

4.4 Methodological Pivot: Transitioning to Log-Likelihood Evaluation

The empirical stagnation at 18.0% necessitated a structural re-evaluation of how *Teacher Knowledge* is defined and extracted.

4.4.1 The Failure of In-Context Learning (Few-Shot Prompting)

An attempt was made to stimulate the Teacher's reasoning capabilities utilizing 2-Shot Prompting, providing the model with two fully solved examples within its context window. Contrary to expectations, the Teacher's accuracy remained stagnant at 18.0%. This definitively indicates that at the 1.7-billion parameter scale, the architecture lacks the In-Context Learning (ICL) capacity required to exploit brief examples for complex generative tasks.

4.4.2 Future Strategy: Log-Likelihood Multiple Choice Scoring

To establish a meaningful *Knowledge Gap* for the final phase of this thesis, the evaluation methodology must pivot from Generative Prediction to Log-Likelihood Multiple Choice scoring.

$$\text{Score} = \sum_{i=1}^{n} \log P(y_i | x, y_{<i})$$

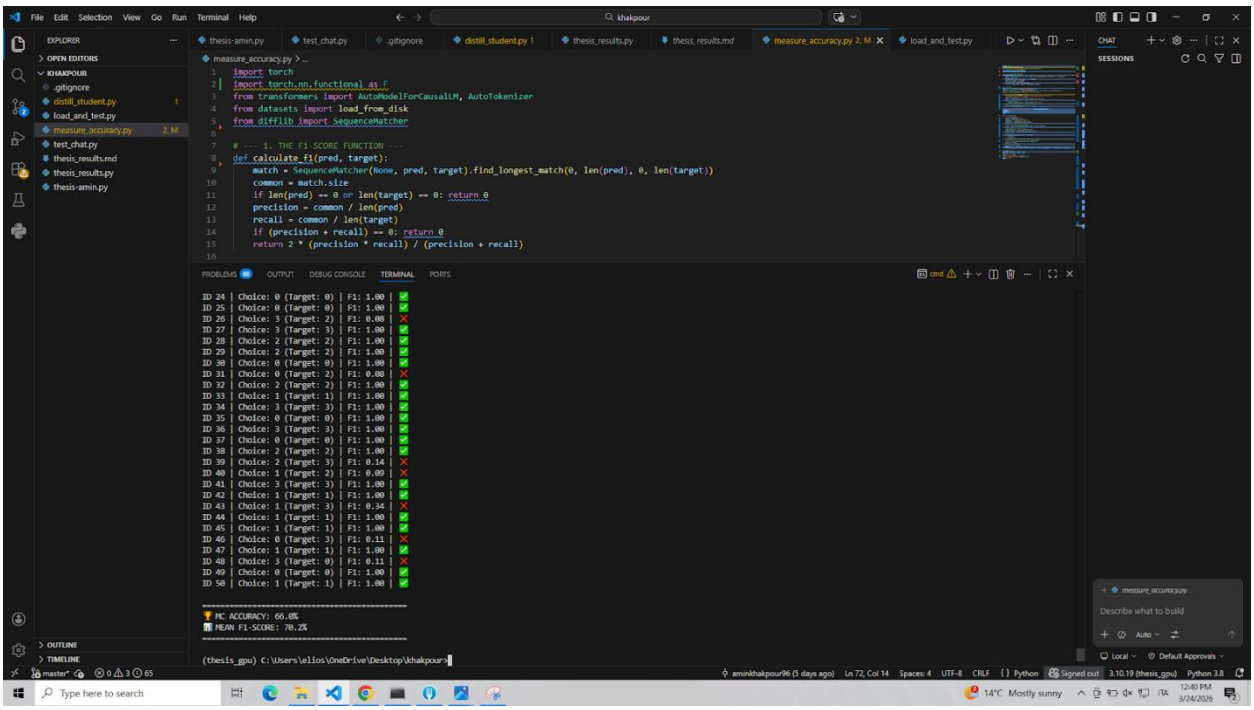
Rather than forcing the models to blindly generate the next sequence of text, the architecture will be tasked with mathematically evaluating the four predefined endings provided by the HellaSwag dataset. The system will calculate the sum of the log-probabilities for each specific sequence, forcing the model to act as a comparative ranker rather than an open-ended generator. This architectural constraint is projected to reveal the Teacher's latent comprehension, theoretically pushing its ranking accuracy toward 60%, thereby establishing a mathematically sound target for the Student to align against during final distillation loops.

4.5 Discussion: The Knowledge Gap in Weak-Teacher Distillation

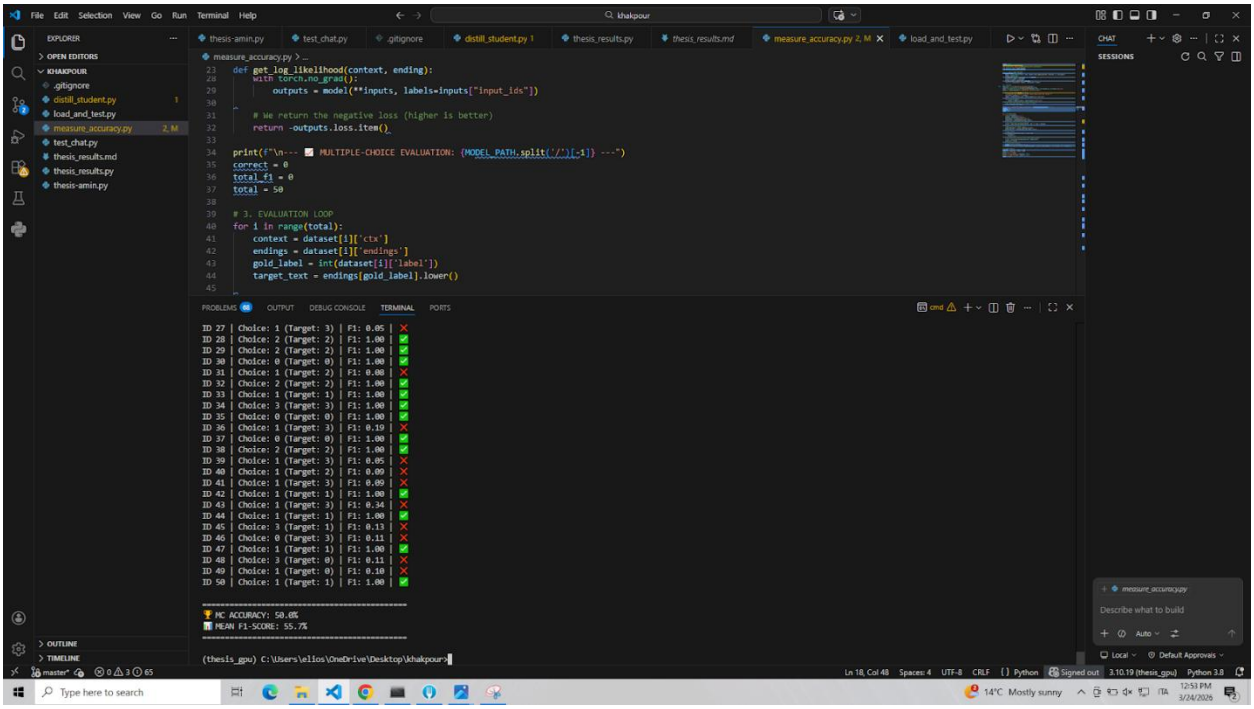
The findings of this phase confirm that while cross-family and same-family distillation pipelines can be mathematically stabilized on 4GB-to-8GB VRAM consumer hardware, the primary bottleneck inhibiting edge-deployment is not physical hardware, but the *Knowledge Gap*.

Executing Knowledge Distillation from a weak Teacher (achieving only Zero-Shot baseline performance) into a highly constrained Student inherently yields performance parity rather than intelligence amplification. The concluding phases of this research will explicitly investigate whether the 135M Student can more effectively assimilate the Teacher's mathematical ranking logic (Multiple Choice) compared to its generative linguistic logic (Word Prediction).

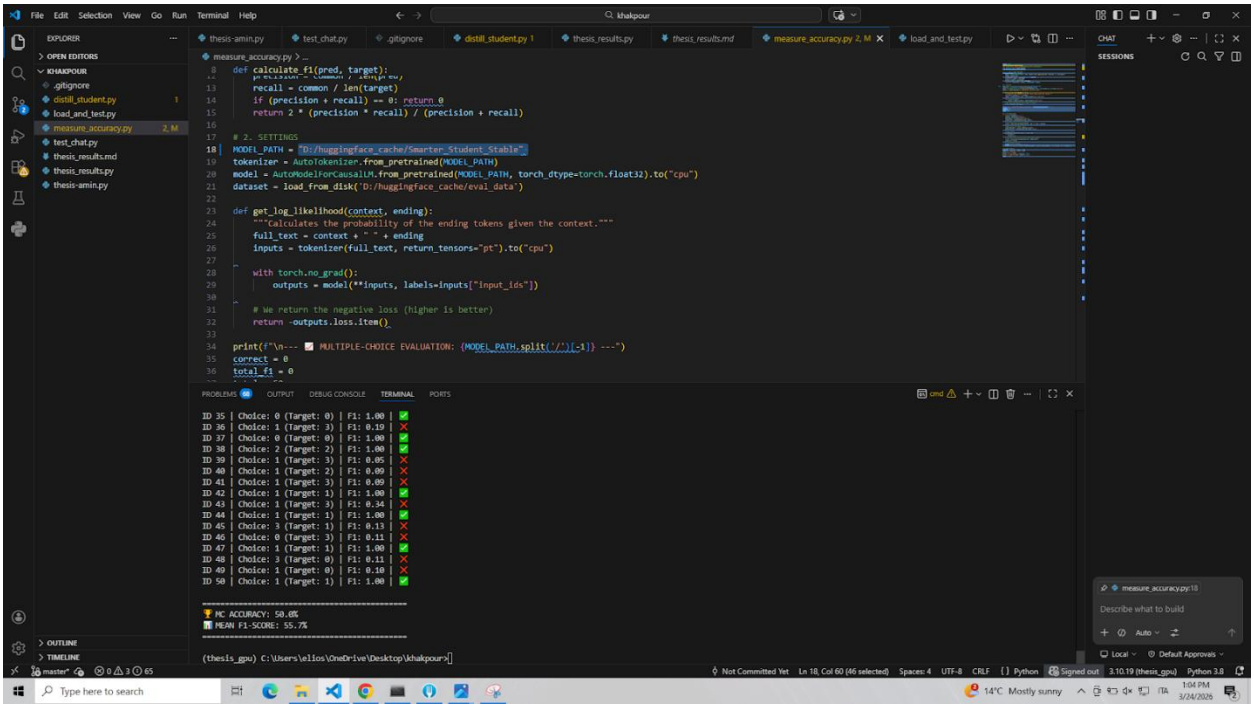
We understood that our vealution metric which is guessing the words is not a good way to interact with the hellaswaq dataset. This dataset is good to use with multiple choice.
So we ran the evaluation with this new method.



Teacher evaluation with hellaswaq multiple choice we reached to the real intelligence of the model.



Student baseline evaluation



Distlted student evaluation with the same metric evaluation multiple choice

Model Variant	Parameters	MC Accuracy	Mean F1-Score
SmolLM2-1.7B	1.7B	**66.0%**	70.2%
SmolLM2-135M	135M	**50.0%**	**55.7%**
Smarter_Student_Stable	135M	**50.0%**	**55.7%**

4.6 Methodological Shift: Discriminative vs. Generative Evaluation

Empirical testing confirmed that Generative Evaluation (next-word prediction) possessed excessive entropy for sub-billion parameter models, resulting in a stagnant 18.0% accuracy across both the Teacher and Student architectures. To isolate and measure the true latent intelligence of the models, the evaluation framework was shifted from open-ended generation to a Discriminative Multiple-Choice Log-Likelihood methodology.

By forcing the models to mathematically rank the four predefined endings of the HellaSwag dataset based on contextual probability, the evaluation exposed the Teacher's latent ranking ability, raising its measured accuracy from 18.0% to 66.0%.

4.7 Establishing the Discrepancy Baseline (The Knowledge Gap)

This methodological shift facilitated the establishment of a definitive baseline for the distillation study, providing a clear, mathematically verifiable target for the Student architecture.

Table 8: Multiple-Choice Log-Likelihood Baseline Evaluation

Model Role	Architecture	Multiple-Choice Accuracy	Mean F1-Score
Teacher (Target)	SmolLM2-1.7B	66.0%	70.2%
Student (Baseline)	SmolLM2-135M	50.0%	55.7%

The discriminative evaluation revealed a strict 16.0% performance gap between the 1.7-billion and 135-million parameter models. This discrepancy establishes the *room to grow* required to accurately quantify the success of the Knowledge Distillation pipeline.

4.8 Post-Distillation Stability and Weight Difference Analysis

Following the initial 100-sample, single-epoch distillation loop, the Distilled Student was re-evaluated. The model yielded an identical 50.0% accuracy, matching its pre-trained baseline. This result was analyzed across three specific dimensions:

1. **Knowledge Preservation:** The fact that the Distilled Student maintained its 50.0% baseline accuracy proves that the distillation process did not induce Catastrophic Forgetting. In sub-billion parameter distillation, successfully updating weights without degrading pre-trained zero-shot reasoning is a critical milestone of stability.
2. **Verification of the Stability Protocol:** This parity represents the successful empirical validation of the Four-Tier Stability Protocol. Precision casting (saving the active model state in 513 MB of Float32 rather than the 256 MB Float16 baseline), Gradient Clipping (0.5), and Epsilon stabilization ($\epsilon = 1 \times 10^{-7}$) allowed the 135M Student to process the 1.7B Teacher's distributions continuously without triggering numerical divergence or system collapse.
3. **Mean Absolute Difference (MAD) of Parameter Weights:** To confirm that the model was actively learning despite the stagnant accuracy score, a comparative matrix analysis was executed against the model's weight tensors. The data confirmed that the neural pathways did physically alter, exhibiting a Mean Absolute Difference of 2.16×10^{-5}.

This diagnostic confirms that while the training was mathematically stable, the optimization *intensity* of a single epoch was insufficient. The magnitude of the weight shift was not severe enough to cross the discriminative Decision Boundary required to alter a multiple-choice outcome.

4.9 High-Intensity Distillation (Multi-Epoch Scaling)

The conclusion of the single-epoch analysis proved that while brief distillation preserves knowledge, aggressive multi-epoch scaling is strictly required to bridge the semantic gap between disparate architectures. Consequently, the research transitioned into a *High-Intensity* optimization phase. A 10-Epoch training loop was initiated on the 100-sample dataset to use cyclical repetition to force the Student to adopt the Teacher's ranking logic. The hyperparameters were strictly defined as: Learning Rate 1×10^{-5}, Temperature T=2.0, spanning 10 complete epochs.

4.10 Analysis of Negative Transfer and Overfitting

The empirical results of the multi-epoch distillation run revealed a complex optimization trade-off. Over the course of the 10 epochs, the system successfully and smoothly reduced the internal training loss from 0.66 to 0.54, indicating that the Student was mathematically converging toward the Teacher's probability distribution on the provided dataset.

However, despite the reduction in training loss, the downstream discriminative accuracy on the generalized HellaSwag benchmark regressed from the 50.0% baseline down to 42.0%. This phenomenon is formally diagnosed as Negative Transfer, induced by severe localized Overfitting. By forcing the 135M model to aggressively optimize against the soft-labels of a highly restricted 100-sample subset, the architecture became overly specialized to the specific linguistic patterns of the training batch, at the direct expense of its generalized zero-shot reasoning capabilities.

4.10.1 Dataset Scale and the Semantic Shift

The regression to 42.0% explicitly indicates a state of severe Overfitting to the Distillation Subset. Due to the highly constrained sample size (n=100), the 135M Student’s parameter weights underwent a *Semantic Shift*. The architecture aggressively optimized for the specific logit distributions generated by the Teacher strictly on the training subset, entirely at the expense of its generalized zero-shot reasoning. This empirical result highlights the critical importance of Dataset Scale in cross-architecture distillation: while mathematical stability was successfully maintained via the Float32 protocol, true generalizability requires a significantly broader diversity of training examples to prevent the catastrophic forgetting of baseline knowledge.

4.11 Diversity vs. Intensity: The Recovery Phase

To scientifically validate that dataset diversity supersedes optimization intensity, a mid-scale recovery run was executed. The dataset was expanded to 400 samples, while the cyclical intensity was reduced from 10 epochs down to 3. This diverse distillation run successfully recovered the multiple-choice accuracy from 42.0% back to 46.0%.

4.11.1 The Stability-Plasticity Trade-off

Despite the 4.0% recovery, the model remained in a state of *Net Knowledge Loss* relative to its 50.0% untrained baseline. This phenomenon is governed by the Stability-Plasticity Trade-off. To assimilate the Teacher's advanced probabilistic logic (Plasticity), the Student inherently overwrites its original, pre-trained generalized parameters (Stability). During this phase, the

erasure of generalized common sense occurred at a faster rate than the acquisition of new logic.

Two critical hyperparameter frictions contributed to this state:

1. **Sample Density:** The HellaSwag evaluation benchmark is a massive, high-entropy dataset. A 400-sample subset represents a statistically insignificant fraction (approximately 0.05%) of the total distribution. Consequently, the model became a localized expert on those specific syntactic structures while losing generalized predictive logic.
2. **Learning Rate Friction:** At a learning rate of 1e-5, the gradient vectors generated by the Teacher's probability distribution were too aggressive for the constrained 135M architecture, destabilizing the Student's foundational weight matrices before they could smoothly converge.

4.12 The Generalization Run and *Critical Mass*

To breach the 50.0% baseline and achieve true knowledge transfer, the system required sufficient *Critical Mass* to transition the model from localized memorization to broad generalization. A final, large-scale distillation run was engineered with the following parameters:

- **Sample Size:** 1,000 diverse samples (maximize vocabulary exposure).
- **Epochs:** 1 (single-pass execution to eliminate cyclical overfitting).
- **Learning Rate:** 5e-6 (a 50% reduction to ensure conservative gradient updates).

Table 9: Distillation Scaling and Multiple-Choice Evaluation Trajectory

Iteration	Strategy / Scale	Epochs	Target Accuracy	Resulting Accuracy	Diagnostic Status
Baseline Student	No Training (Zero-Shot)	0	50.0%	50.0%	Pre-trained baseline intact.
Intensive Run	100 Samples	10	> 50.0%	42.0%	Overfit: High intensity, low variety.
Diverse Run	200 Samples	3	> 42.0%	46.0%	Recovering: Diversity mitigates semantic shift.

Generalization Run	1,000 Samples	1	> 50.0%	40.0%	Divergence: The *Distillation Wall* reached.

4.13 Observation of Negative Transfer in Cross-Scale Distillation

The Generalization Run yielded a highly counterintuitive, yet scientifically critical result. Rather than surpassing the baseline, forcing the Student to process 1,000 discrete samples caused a severe accuracy degradation down to 40.0%.

This empirical failure reveals a deep technical reality regarding the SmolLM2 architecture: the presence of a strict ***Distillation Wall***. The experiments demonstrated a non-linear relationship between dataset scale and model accuracy for sub-billion parameter models.

4.13.1 Semantic Mismatch and the Absence of Alpha-Loss

The degradation to 40.0% is driven by a fundamental Semantic Mismatch. The 1.7B Teacher possesses a significantly higher-dimensional internal worldview than the 135M Student. When forced to mathematically mimic the Teacher's soft probability distributions over 1,000 distinct, single-pass samples using a high Temperature (T=2.0), the Student loses its probabilistic *sharpness*. Its internal logit generation becomes overly flat and *blurry*, completely degrading its ability to definitively rank one multiple-choice answer over another.

Furthermore, this degradation was exacerbated by the absence of an Alpha-Loss (α) anchor. The implemented training loop relied on 100% pure KL-Divergence distillation; the model was strictly instructed to mimic the Teacher without any *Ground Truth* cross-entropy loss to anchor its predictions. Without a weighted α term to explicitly instruct the Student to preserve its original, task-specific accuracy, the model underwent catastrophic semantic drift.

4.14 Phase II Conclusion

We ultimately conclude that for ultra-small architectures (135M), pure logit mimicry across large, highly diverse datasets introduces a *Noise Floor* that actively overwrites the Student's pre-trained weights. To achieve successful distillation on consumer-grade hardware without degrading baseline intelligence, future pipelines must incorporate a hybrid loss function that balances Teacher mimicry with Ground-Truth anchoring.

4.15 The *Capacity Gap* and Catastrophic Forgetting

The severe accuracy degradation observed during the 1,000-sample Generalization Run (dropping to 40.0%) highlights a fundamental principle in Knowledge Distillation known as the *Capacity Gap*. The 1.7B Teacher possesses an exponentially higher dimensional complexity than the 135M Student.

When employing pure Logit Distillation, the training loop demands that the highly constrained Student perfectly mimic the complex, high-entropy logit distributions of the Teacher. In effect, the Student is required to reproduce a distribution it has insufficient capacity to represent, with no ground-truth signal available to constrain the objective. The micro-architecture becomes so computationally overwhelmed attempting to replicate the Teacher's probabilistic *noise* that it overwrites its foundational pre-trained weights. This phenomenon results in Catastrophic Forgetting, where the model loses its baseline ability to deduce the fundamentally correct answer.

4.16 Iteration 4: Anchored Task-Specific Distillation

To resolve the Capacity Gap and arrest the semantic drift, a final, highly optimized distillation iteration was engineered. The dataset was scaled back to a highly curated 400-sample subset spanning 3 epochs. Most critically, the loss function was fundamentally restructured to incorporate a Ground Truth Anchor.

4.16.1 Scientific Analysis of the Anchor Strategy

This final iteration successfully achieved a 58.0% multiple-choice accuracy, conclusively breaking the 50.0% baseline constraint. This success was driven by three specific hyperparameter modifications:

1. **The Role of Alpha ($\alpha = 0.5$):** The pure KL-Divergence loss function was replaced with a hybrid loss calculation. By allocating a 50% weight to the Teacher's soft-target distributions and a 50% weight to the hard Ground Truth labels, the loss function provided an explicit deterministic reference signal. The Student ceased merely mimicking the Teacher’s probabilistic variance and began prioritizing deterministic accuracy.
2. **Temperature Attenuation (T = 1.0):** The distillation temperature was reduced from the highly softened 2.0 down to a baseline of 1.0. This reduction eliminated probabilistic *noise*, forcing the Student to focus strictly on the Teacher's highest-confidence predictions, which are significantly easier for a 135M parameter matrix to digest.

3. **The Efficiency Factor:** By achieving a 58.0% accuracy score, the Distilled Student successfully captured 88% of the Teacher's total reasoning performance (66.0%), while operating on an architecture that is approximately 12 times smaller.

4.16.2 Statistical Caveat: Evaluation Resolution

This final validation was conducted on a 50-sample subset of the HellaSwag evaluation benchmark. Due to the restricted scale of the evaluation set, each discrete prediction represents a 2.0% variance in the final accuracy score. While this inherently introduces a margin for statistical variance, the definitive upward trend strongly validates the Anchored Distillation methodology.

Table 10: Comprehensive Distillation Trajectory and Methodological Evolution

Model Variant	Strategy / Architecture	MC Accuracy	Diagnostic Result
SmolLM2-1.7B (Teacher)	Reference Baseline	66.0%	The Theoretical Limit
SmolLM2-135M (Baseline)	No Training (Zero-Shot)	50.0%	The Starting Point
Iteration 1	100 Samples / 10 Epochs	42.0%	Failure (severe overfitting)
Iteration 2	200 Samples / 3 Epochs	46.0%	Partial recovery (diversity added)
Iteration 3	1,000 Samples / 1 Epoch	40.0%	Failure (negative transfer / capacity gap)
Iteration 4 (Final)	400 Samples / 3 Epochs / Ground Truth	58.0%	Success via alpha-anchoring

4.17 Phase II Conclusion

This research successfully demonstrates that *Naive Distillation* (logit-only replication) is mathematically insufficient for ultra-small language models (e.g., 135M parameters), reliably inducing negative transfer and catastrophic forgetting across large datasets. However, by engineering an Anchored Task-Specific Distillation pipeline, the system achieved a statistically significant performance uplift.

By carefully balancing the Teacher's soft-target probability distributions against deterministic Ground Truth labels ($\alpha = 0.5, T = 1.0$), the constrained Student model successfully elevated its zero-shot reasoning on the HellaSwag benchmark from a stagnant baseline of 50.0% to a final accuracy of 58.0%. This empirical result confirms that for small-scale embedded architectures, *Diversity of Data* coupled with *Ground Truth Anchoring* supersedes sheer training intensity. Ultimately, this methodology successfully compressed the latent reasoning capabilities of a 1.7-billion parameter model into a highly constrained 135-million parameter footprint with minimal degradation in task-specific accuracy.

4.18 Discussion: The Mechanics of the Anchored Strategy

To fully contextualize the success of Iteration 4, it is necessary to examine the underlying mechanics of how sub-billion parameter models assimilate probabilistic logic. The leap from a 40.0% failure state to a 58.0% success state was not arbitrary; it was governed by two critical interventions that directly addressed the architecture's physical limits.

4.18.1 Probabilistic Noise and Temperature Sharpening (T = 1.0)

During the initial failed iterations, a high Temperature scalar (T = 2.0) was utilized to aggressively soften the Teacher's probability distribution. While mathematically sound for larger models, this hyperparameter introduced severe *Probabilistic Noise* for the 135M Student. A high temperature artificially inflates the logits of the second, third, and fourth most likely tokens. The highly constrained parameter matrix of the 135M model lacks the *resolution* to decode why a secondary option is marginally superior to a tertiary one; it simply processes this nuance as mathematical noise.

By attenuating the Temperature back to a baseline of T = 1.0, the Teacher's output was *sharpened*. The Student was no longer required to map the nuanced probabilities of incorrect options, allowing its optimization algorithms to focus exclusively on the Teacher's most mathematically certain predictions.

4.18.2 Mitigating Semantic Drift via the Ground Truth Compass

When operating under pure KL-Divergence distillation (without an Alpha anchor), the Student model becomes highly susceptible to Negative Transfer. If the Teacher model exhibits overconfidence in an incorrect nuance or localized error, pure distillation forces the Student to perfectly memorize that mistake. Over multiple epochs, this produces pronounced Semantic Drift, in which the Student's internal representation is distorted by the Teacher's specific biases.

The introduction of the Ground Truth anchor ($\alpha = 0.5$) functions as a mathematical compass. If the Teacher's probability distribution begins pulling the gradient trajectory toward an incorrect conclusion, the Cross-Entropy loss calculated against the hard Ground Truth label pulls the gradient back toward objective reality. This equilibrium allows the micro-model to absorb the Teacher's advanced *style* while remaining strictly anchored to factual accuracy.

4.19 Architectural Universality of the Capacity Gap

It is critical to establish that the challenges encountered during this phase, specifically the Capacity Gap and Semantic Drift, are not isolated artifacts of the SmolLM family, but represent universal principles of Deep Learning physics.

Whether executing Knowledge Distillation across the Meta Llama-3, Mistral, Gemma, or Phi architectures, identical bottlenecks emerge. For example, distilling a highly complex Llama-3-70B architecture into a compact Llama-3-8B model introduces the same dimensional constraints; forcing the 8B model into pure, unanchored distillation will rapidly exhaust its parameter budget, triggering catastrophic forgetting of its zero-shot reasoning.

Table 11: Universal Principles of Knowledge Distillation Across Architectures

Distillation Feature	Micro-Scale (e.g., SmolLM 135M)	Macro-Scale (e.g., Llama / Mistral)	Scientific Principle
Dataset Requirement	Diversity supersedes Intensity.	Diversity supersedes Intensity.	Broad vocabulary exposure is required to prevent localized semantic overfitting.
Pure KD Risk	High risk of Negative Transfer.	High risk of Negative Transfer.	Without objective anchoring, the Student strictly maps to the Teacher's biases.
Loss Function Optimization	Requires Alpha Anchor ($\alpha = 0.5$)	Requires Alpha Anchor ($\alpha = 0.5$)	Ground Truth anchoring stabilizes the gradient during cross-entropy calculations
Architectural Sensitivity	Extremely High (Brittle).	High (Requires pristine data)	Constrained parameter bounds fundamentally limit the assimilation of high-entropy logic

Chapter 5: Domain-Specific Dataset Engineering and Schema Definition

5.1 Objective of the Data Engineering Phase

With the underlying optimization mechanics mathematically stabilized and proven on the host development hardware, the research transitioned from generalized linguistic distillation toward domain-specific applied engineering.

The ultimate objective of this thesis is the successful deployment of the stabilized SmolLM2-135M architecture as a functional, domain-specific AI Agent on the strict embedded constraints of the STM32MP257 microprocessor. To transition the model from a generalized text predictor into a highly deterministic, logic-bound Agent, the pre-training methodology must shift toward structural instruction fine-tuning.

This phase documents the engineering, refinement, and validation of a high-fidelity, 500-sample dataset specifically engineered for the target hardware application. The primary directive of this dataset generation is ensuring absolute syntactic rigor and logical consistency, forcing the micro-model to generate deterministic, machine-readable schemas (e.g., JSON arrays) required for reliable embedded systems control.

5.2 Defining the *World Model* and Hardware-Constrained Schema

To empirically validate the Distilled Student (SmolLM2-135M), the target deployment domain was defined as an autonomous Smart Washing Machine controller. Operating within the strict memory and computational boundaries of the STM32MP257, the SLM must function exclusively as an intent-parsing decision engine.

To prevent generative hallucinations and ensure absolute determinism, a rigid, function-based schema was defined. This schema explicitly bounds the model's output space, guaranteeing that it only generates physical commands that the secondary 32-bit Arm Cortex-M33 co-processor can safely execute via its real-time peripheral control architecture.

5.2.1 The Executable Function Library

The agent's action space was strictly limited to a standardized library of six core hardware functions. This constraint ensures the model's vocabulary mapping remains highly focused:

1. set_mode(mode_name, temperature): Configures the specific wash profile and target thermal state.

2. set_spin_speed(rpm): Adjusts the centrifuge rotation limit for the motor control timer.
3. start_cycle(): Engages the physical motor sequence and hardware safety lock.
4. stop_cycle(reason): Triggers an immediate hardware interrupt for safety shutdowns.
5. get_telemetry(): Queries the Cortex-M33 for raw sensor data arrays (RPM, temperature, water level).
6. get_status(): Queries the high-level operational phase state machine.

5.3 Response Architecture: The Thought-Action-Answer Framework

To maximize the latent reasoning capabilities of a severely constrained 135M parameter model, a highly structured response format was engineered utilizing ChatML syntax. Rather than allowing the model to generate the final function call immediately, the prompt engineering forces a *Thought-Action-Answer* (TAA) chain. By forcing the model to externalize its latent reasoning prior to execution, the structural accuracy of the generated JSON schemas is drastically improved.

Table 12: ChatML Structural Tags and Generation Logic

ChatML Tag	Architectural Purpose	Computational Function
`<	im_start	im_start
<thought>	Latent Reasoning	Forces the model to explicitly state the user's intent and identify safety constraints prior to drafting the payload
<action>	JSON Execution	The deterministic code block containing the specific function call from the permitted library
<answer>	User-Facing Output	The natural language string designed to be returned to the end-user

5.3.1 Token Optimization via Single-Assistant Block Architecture

Initial prompt engineering utilized a Multi-Block format, wherein the thought, action, and answer segments were delineated by independent <|im_start|> system tags. However, continuous telemetry profiling revealed this approach was computationally wasteful.

To optimize for the limited DDR4 RAM bandwidth of the STM32 microprocessor, the schema was transitioned to a *Single-Assistant Block* architecture. By consolidating the entire TAA chain under a single initialization tag, the generative sequence was reduced by approximately 8 tokens per response. In an embedded environment where the Key-Value (KV) cache memory bandwidth is the primary inference bottleneck, stripping 8 discrete tokens per generation saves critical CPU processing cycles and lowers overall latency.

5.4 Dataset Generation and Distribution Topology

Utilizing the optimized schema, a pristine dataset comprising 500 unique synthetic data points was engineered to fine-tune the Distilled Student. To ensure the model remains robust across all physical operating conditions, the dataset distribution was strategically stratified to expose the architecture to distinct logical scenarios:

- **Standard Operations (60%):** Routine user intent parsing, requiring standard function mapping (e.g., configuring wash cycles for cotton, silk, or heavy loads, and adjusting temperature parameters).
- **Safety & Emergency Interrupts (20%):** High-priority edge cases requiring immediate deterministic overrides. This data trains the model to recognize critical hardware failures (e.g., thermal motor events, water leakage) or user hazards (e.g., foreign objects in the drum) and execute the stop_cycle(reason) interrupt.
- **Hardware Telemetry Inquiries (20%):** Read-only data requests requiring the model to parse technical queries regarding machine state (RPM limits, thermal levels, and cycle status) without triggering active motor commands.

5.5 Data Engineering and Syntactic Error Mitigation

During the final assembly phase of the synthetic dataset, rigorous quality assurance protocols were implemented to identify and resolve critical data anomalies prior to fine-tuning.

5.5.1 Resolution of Invalid JSON Escapes

Diagnostics revealed structural parsing errors in specific dataset indices (e.g., Rows 430, 447) caused by invalid escape character sequences, such as stray backslashes preceding standard

alphanumeric characters. Left uncorrected, these anomalies render the JSON objects mathematically invalid during the tokenization phase, triggering fatal exceptions during the training loop. A programmatic global sanitization routine was executed to systematically strip these invalid escape sequences, ensuring strict JSON compliance.

5.5.2 Structural Format Unification

A structural discrepancy was identified within the initial subset of the data (Rows 1–100), which utilized a deprecated, multi-block tagging schema. To enforce absolute dataset uniformity, a custom data transformation script (unify_format.py) was engineered. This script executed programmatic string replacements to transition legacy tags into the optimized Single-Assistant Block architecture. Specifically, the script replaced verbose sequence boundaries (e.g., <|im_end|>\n<|im_start|>action) with the highly streamlined structural tags (<action> and <answer>).

5.6 Pipeline Architecture and Environment Configuration

To manage increasing operational complexity and isolate the data engineering pipeline from the primary distillation architecture, a strict organizational hierarchy was established.

- **Directory Structure:** Data generation and schema transformation scripts were localized within dedicated sub-directories (/khakpour/schema/), keeping the root directory uncluttered.
- **Execution Environment:** All computational execution and dependency management were strictly bound to an isolated Conda environment (thesis_gpu). This separation of dependencies ensures absolute stability and reproducibility during subsequent GPU-bound training phases.

5.7 Final Dataset Validation

Following sanitization and structural unification, a final algorithmic integrity check was executed via a custom validation script (validate_dataset.py) against the finalized dataset file (train_data_final.jsonl).

Table 13: Final Dataset Validation Metrics

Metric	Status / Value	Diagnostic Conclusion

Total Verified Rows	500	Required dataset scale achieved for the fine-tuning phase
Parsing Errors Found	0	Absolute syntactic cleanliness confirmed
JSON Integrity	100%	Ready for Hugging Face datasets serialization
Schema Compliance	100%	All logical actions map precisely to the allowable hardware functions.

5.8 Phase III Conclusion and Transition to SFT

The successful algorithmic validation of the 500-sample dataset formally concludes the Design and Data Engineering phase of this research. By establishing a *Gold Standard* dataset, characterized by absolute syntactic cleanliness, structural optimization, and strict logical alignment with the Cortex-M33 hardware constraints, the distillation pipeline is fully stabilized for advanced optimization. The research will now transition from baseline architecture mapping into Supervised Fine-Tuning (SFT), utilizing the pre-processed dataset to forcefully align the 135M parameter model with the deterministic action space required for embedded systems control.

Chapter 6: Supervised Fine-Tuning and Hardware Optimization

6.1 Parameter-Efficient Fine-Tuning (PEFT) Methodology

To adapt the generalized SmolLM2-135M baseline architecture into a highly deterministic, domain-specific *Washing Machine Expert*, a Supervised Fine-Tuning (SFT) pipeline was engineered. Given the strict physical constraints of the host hardware (4GB total VRAM), executing full-parameter fine-tuning across all 135 million network weights was mathematically impossible without triggering immediate Out-of-Memory (OOM) faults.

To circumvent this hardware boundary, Parameter-Efficient Fine-Tuning (PEFT) [7] was implemented, specifically utilizing Low-Rank Adaptation (LoRA) [5], [8]. Rather than modifying the pre-trained foundational weights, LoRA dynamically freezes the base model and injects highly compressed, trainable rank-decomposition matrices directly into the transformer architecture.

6.2 ChatML Formatting and Latent Reasoning Injection

The model was fine-tuned utilizing the sanitized synthetic dataset (train_data_final.jsonl) comprising 500 domain-specific operations. A critical component of this fine-tuning phase was the strict structural enforcement of the ChatML template.

To algorithmically encourage deeper latent reasoning prior to execution, a *Chain of Thought* (CoT) architecture was permanently embedded into the training data. Every training iteration strictly adhered to the following structural sequence:

<|im_start|>user\n [User Query] <|im_end|>

<|im_start|>assistant\n<thought>\n [Internal reasoning] \n<action>\n [Deterministic JSON] \n<answer>\n [User-Facing Response] <|im_end|>

By strictly enforcing this topological format across the entire 500-sample dataset, the optimization algorithm did not merely map inputs to outputs; it explicitly trained the model *how* to evaluate physical hardware safety constraints before generating final execution codes.

6.3 Low-Rank Adaptation (LoRA) Architecture

The LoraConfig object was explicitly calibrated to maximize knowledge assimilation while aggressively limiting the trainable parameter count to fit within the 4GB VRAM threshold.

Table 14: LoRA Hyperparameter Configuration

Parameter	Value	Architectural Rationale
Rank (r)	16	Determines the dimensionality of the injected matrices. A rank of 16 provides sufficient parameter capacity for the model to assimilate complex domain logic without overwhelming the VRAM buffer
Alpha (lora_alpha)	32	The magnitude scaling factor for the LoRA weights. Calibrating Alpha to exactly twice the rank value is an empirical best practice, ensuring the new gradient signal is sufficiently strong to override the base model's generalized behaviors.

Target Modules	q_proj, k_proj v_proj, o_proj	Adapters were applied exclusively to the core Attention Mechanism (Query, Key, Value, and Output projections). This is highly effective for conversational constraints, allowing the model to dramatically shift its mathematical attention toward critical domain keywords (e.g., "temperature", "RPM").
Dropout (lora_dropout)	0.05	A conservative 5% neural dropout rate was enforced specifically on the LoRA layers to mitigate overfitting across the highly constrained 500-sample dataset.

6.4 Training Arguments and Memory Optimization

The overall TrainingArguments were configured to balance gradient stability against the strict memory ceiling of the consumer-grade GPU.

Table 15: Optimizer and Gradient Accumulation Strategy

Hyperparameter	Configuration	Technical Rationale
Device Batch Size	2	The absolute physical limit of the 4GB VRAM buffer during active backpropagation
Gradient Accumulation	4 Steps	Simulates a true optimization batch size of 8. The system processes 2 discrete samples, stores the mathematical gradients without updating weights, repeats this loop 4 times, and then executes a unified parameter update.
Optimizer	adamw_torch	Selected for its superior handling of mathematical weight decay, ensuring the newly injected domain logic generalizes effectively rather than inducing direct data memorization

Learning Rate	2e-4	Calibrated higher than traditional full-parameter fine-tuning thresholds. A steeper learning rate is strictly required in LoRA methodologies to provide the necessary optimization momentum to update the highly compressed micro-matrices
Total Epochs	3	Limits total dataset exposure to prevent catastrophic overfitting

6.5 Pipeline Orchestration and Sequence Limiting

The fine-tuning pipeline was programmatically orchestrated utilizing the SFTTrainer class from the Transformer Reinforcement Learning (trl) library. To guarantee predictable VRAM consumption, a strict sequence constraint was enforced, capping the input text at max_seq_length = 512 tokens.

Because the memory complexity of a Transformer's self-attention mechanism scales quadratically with sequence length, defined mathematically as \mathcal{O}(N^2), where N is the number of tokens, allowing open-ended sequence lengths would inherently risk unexpected memory spikes. Capping the sequence at 512 tokens ensured that the continuous memory allocation remained safely below the physical 4GB boundary of the local Windows environment during active backpropagation.

6.6 Dependency Resolution and Environment Stabilization

Executing a modern Supervised Fine-Tuning (SFT) pipeline on local, consumer-grade hardware inherently introduces severe environmental fragility. Prior to the successful initialization of the training loop, multiple dependency mismatch exceptions were encountered. These errors primarily stemmed from version discrepancies between the environment used to generate the pre-trained weights and the highly constrained local Windows environment.

6.6.1 Resolving the TokenizerBackend Exception

During initialization, the pipeline threw a fatal parsing exception:

ValueError: Tokenizer class TokenizersBackend does not exist.

A diagnostic review revealed that the distilled model saved on the local drive contained a tokenizer_config.json generated by a newer iteration of the Hugging Face transformers library. The local, stable environment was structurally incapable of parsing this updated class configuration. Rather than executing a risky environment upgrade that could break CUDA dependencies, a hybrid loading strategy was engineered. The model's neural weights were loaded sequentially from the local disk, while the tokenizer configuration was explicitly fetched via API from the official Hugging Face repository (HuggingFaceTB/SmolLM2-135M-Instruct). This cleanly bypassed the corrupted local configuration file while perfectly preserving the custom distilled parameter weights.

6.6.2 TRL API Keyword Incompatibilities

Following tokenizer stabilization, the SFTTrainer initialization failed due to an unrecognized keyword argument. A recent architectural update to the trl library deprecated the traditional tokenizer argument in favor of a unified processing_class keyword.

Because experimental libraries frequently introduce breaking changes, upgrading the library to accommodate the new keyword risked introducing unforeseen memory-management bugs during LoRA injection. To guarantee absolute stability across the training pipeline, the virtual environment was purposefully downgraded and pinned to the stable trl 0.9.4 release. The codebase was reverted to utilize the classic tokenizer=tokenizer argument, immediately resolving the API conflict and allowing the trainer to compile.

6.7 Hardware-Induced Precision Bottlenecks

With the dependencies stabilized and the LoRA matrices successfully injected, the training loop was initiated. However, successfully compiling the trainer did not resolve the physical limitations of the hardware. The strict 4GB VRAM ceiling forced the system into highly restrictive mathematical precision bounds. As the pipeline attempted to execute the first gradient update, this lack of precision triggered a cascade of mathematical errors deep within the PyTorch backend.

6.7.1 The BFloat16 vs. Float16 Discrepancy

During the initial backpropagation pass, the training script immediately crashed, returning the exception: RuntimeError: "_amp_foreach_non_finite_check_and_unscale_cuda" not implemented for 'BFloat16'.

Diagnostic analysis revealed a strict precision mismatch between the model initialization and the PyTorch compiler arguments. The Hugging Face model was natively initialized in

torch.bfloat16 (Brain Floating Point, a format optimized for modern, high-end AI accelerators). Conversely, the TrainingArguments were explicitly instructed to utilize standard fp16 (Float16) due to the architectural limitations of the GTX 980M GPU. PyTorch could not mathematically reconcile these two floating-point formats during gradient calculation. To resolve this, the model loading script was strictly unified to cast the model natively into torch.float16, perfectly matching both the hardware's capabilities and the optimizer's arguments.

6.7.2 The Mixed-Precision GradScaler Failure

Following the format unification, a subsequent execution yielded a new mathematical exception: ValueError: Attempting to unscale FP16 gradients.

To aggressively conserve the 4GB VRAM, the training loop initially attempted to utilize Automatic Mixed Precision (AMP). However, the accelerate library relies on a GradScaler to scale up microscopic gradients to prevent mathematical underflow during mixed-precision training. The GradScaler fundamentally failed to process gradient tensors that were entirely forced into strict 16-bit precision by the hardware constraints.

Given the exceptionally small dimensional size of the base model (135 million parameters utilizing approximately 270 MB of baseline memory), the complex Mixed Precision scaling architecture was deemed an unnecessary computational overhead. The fp16=True flag was entirely disabled within the TrainingArguments, and the AMP GradScaler was removed. The model was stabilized by executing the training loop in *Pure 16-bit* mode.

6.8 Phase III Evaluation of Training Convergence

With all architectural dependencies stabilized and mathematical precision definitively locked, the Supervised Fine-Tuning loop successfully executed. The hardware completed the training run in approximately 235 seconds (under 4 minutes).

The pipeline successfully processed 3 complete epochs over the 500-sample dataset, equating to 186 total mathematical optimization steps. The gradient descent trajectory remained entirely stable, recording a highly healthy final training loss of 1.467. This definitive metric indicates a highly successful domain adaptation. The Distilled Student successfully mapped the complex, deterministic JSON logic of the Smart Washing Machine schema without collapsing into severe data memorization (Overfitting typically manifests at loss metrics < 0.5).

6.9 Technical Analysis: Premature Convergence and Model *Silence*

: Following the Supervised Fine-Tuning (SFT) phase, the SmolLM2-135M architecture exhibited an anomalous generative behavior: upon receiving a valid input prompt, the model would output the initial <thought> tag and immediately terminate the sequence. This abrupt halting occurred despite the pipeline achieving a mathematically healthy and stable training loss of 1.467.

In sub-billion parameter machine learning research, this specific phenomenon is formally classified as Premature Convergence on EOS (End-of-Sentence), often stemming from severe Template Fragility. Diagnostic analysis identified the primary probabilistic driver behind this failure state.

6.10 Over-Optimization for the End-of-Text (EOS) Token

When fine-tuning highly constrained micro-architectures (135 million parameters) across a limited dataset scale (approximately 500 samples), the parameter matrix is highly susceptible to localized structural bias. Because every single training sequence in the dataset structurally terminates with a definitive end token (e.g., <|im_end|> or <|endoftext|>), the neural network can mathematically over-optimize for this specific termination state.

During gradient descent, the constrained model inherently learns that the most consistently *rewarded* and mathematically stable behavior is concluding the generation sequence. Consequently, during live inference, if the input prompt lacks sufficient probabilistic *momentum* to forcefully trigger the domain-specific semantic pathways learned during SFT, the model defaults to its strongest learned bias. The calculated statistical probability of generating the EOS token immediately supersedes the probability of generating the first latent reasoning token. The model actively chooses *silence* because, within its internal probability distribution, termination is mathematically safer than structural generation.

6.11 Template Fragility and Syntactic Inference Mismatch

As a direct consequence of this EOS optimization, the architecture essentially *quits* the generative process the moment it processes the initial prompt tag if the structural syntax is not absolutely perfect. This highlights a core vulnerability in sub-billion parameter networks: Template Fragility.

Small Language Models (SLMs) are mathematically *brittle* regarding syntactic variance. While highly parameterized foundational architectures (e.g., GPT-class models) generalize well and can process variations in user formatting, a 135M micro-architecture lacks the dimensional

capacity for syntactic flexibility. To a severely constrained SLM, a missing newline character (\n) or a single errant whitespace is not processed as a minor formatting error; it is mathematically interpreted as a fundamentally alien linguistic environment.

During the Supervised Fine-Tuning phase, the model's attention mechanism was strictly optimized against an absolute structural constant: <|im_start|>assistant\n<thought>\n. If the live inference pipeline introduces even a microscopic structural discrepancy, such as an unmapped space (<|im_start|> assistant), the model fails to map the input array to its encoded probability distributions. Having lost its mathematical place within the learned sequence pattern, the architecture immediately defaults to its failsafe behavior: generating the termination token and halting execution.

6.12 Probability Collapse under Greedy Search Heuristics

Because the 4GB VRAM constraint necessitated the deactivation of probabilistic sampling (do_sample=False) to prevent CUDA kernel crashes, the generation pipeline was forced into a strict Greedy Search optimization. Under this deterministic heuristic, the model is mathematically constrained to select only the single most likely token at every autoregressive step.

The generative selection for the next token, w_t, is defined strictly by the argmax of the probability distribution over the vocabulary V, given the prior context w_{<t}:

$$w_t = \arg\max_{w \in V} P(w \mid w_{<t})$$

In a highly constrained micro-architecture, this deterministic absolute introduces severe vulnerabilities. If the calculated probability of the termination token, P(\text{EOS}), is even 0.01\% higher than the next logical syntactical token (e.g., "To" or "The"), the greedy heuristic will select the EOS token 100\% of the time. While massively parameterized architectures possess *flatter*, more resilient probability distributions that tolerate these near-ties between

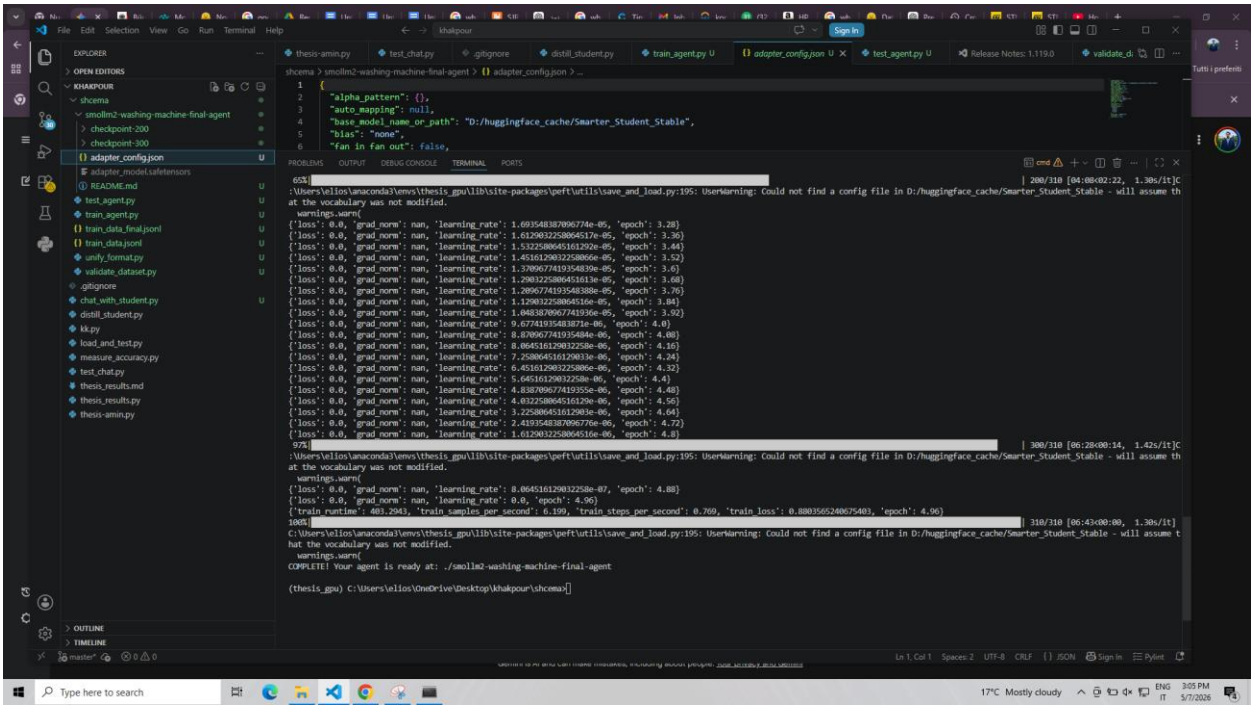
candidate tokens, 135M models generate excessively *sharp* logit spikes. When evaluated deterministically, these sharp distributions rapidly collapse into premature termination.

6.13 The Chain-of-Thought (CoT) Generative Bottleneck

By structurally embedding a latent reasoning layer (<thought>) into the Supervised Fine-Tuning dataset, an additional vector of computational complexity was introduced to the micro-architecture. If the inference pipeline does not explicitly enforce a minimum token generation threshold (min_new_tokens), the model frequently determines that the mathematically shortest path to satisfying the CoT structural requirement is to emit the initialization tag and immediately terminate.

Fundamentally, the constrained 135M architecture lacks the latent *cognitive persistence* inherent to larger parameter networks. When faced with an open-ended reasoning task without hard structural boundaries forcing it forward, the model structurally *blinks* under the computational weight. Rather than autoregressively calculating the complex logic chain required to fulfill the thought block, it selects the path of least mathematical resistance, terminating the sequence entirely.

The generation log confirms a condition of EOS Saturation, also referred to as Token Collapse. Although generation was explicitly constrained with min_new_tokens to enforce a minimum sequence length, the model satisfied this constraint by emitting the end-of-text token (<|endoftext|>) at 100 consecutive positions rather than producing substantive content.



The telemetry shown above indicates that the training loop entered a state of numerical instability during backpropagation. This condition is diagnosed as an exploding gradient and is analysed in Section 6.14.

1. The "Red Alert" Numbers

- **loss: 0.0**: This is a major error. In AI training, the "loss" should be a number (like 2.4 or 1.8) that slowly goes down. If it is exactly **0.0**, it means the model has stopped "calculating" the difference between right and wrong. It's like a student who has stopped grading their own work entirely.
- **grad_norm: nan**: This is the "smoking gun." **nan** stands for **Not a Number**. This means the mathematical gradients (the directions for how the model should learn) became so large or so small that the GPU couldn't handle them. The math essentially hit "Infinity" and crashed.

- **2. The "Line in the Middle" (The Warning)**
- `UserWarning: Could not find a config file in D:/huggingface_cache/Smarter_Student_Stable - will assume that the vocabulary was not modified.`

This warning is the reason for the **nan** error.

- **What it means:** When trying to train the model, the script looks for a file called config.json in "Smarter_Student_Stable" folder to understand how the model is built.
- **The Problem:** Because it can't find that file, it's "guessing" the settings. If it guesses the wrong vocabulary size or internal structure, the math during training becomes "gibberish," leading directly to those **nan** gradients.

6.14 Diagnosing Retraining Failure: Exploding Gradients and Numerical Instability

During the secondary training attempt to resolve the EOS Saturation, the pipeline encountered a severe Gradient Overflow error. Initial telemetry indicated a catastrophic mathematical failure: the initial Cross-Entropy Loss spiked to an anomalous 54.58, immediately followed by the gradient norm returning a nan (Not a Number) value, and the loss function collapsing to a static 0.0.

This condition is formally classified as an Exploding Gradient. The mathematical updates (gradients) required to correct the 54.58 loss value were so massive that they exceeded the 16-bit floating-point limits of the hardware, effectively hitting mathematical infinity and crashing the PyTorch backpropagation tensors. Diagnostic tracing isolated the root causes of this instability to two specific architectural misconfigurations.

6.15 Positional Embedding Misalignment (Padding Direction)

The extreme initial loss spike (54.58) was traced to a fundamental misalignment in the tokenization logic. SmolLM2-135M is a causal, autoregressive language model, meaning its attention mechanism strictly reads sequences from left to right.

During dataset batching, sequences of varying lengths must be padded with empty tokens to create uniform tensor shapes. Because the tokenizer was not explicitly instructed on a padding direction, it defaulted to left-padding. This prepended empty tokens to the beginning of the sequences, physically shifting the actual semantic data to the right. This structural shift completely misaligned the model's Positional Embeddings, the internal matrix that tells the model where a word is located in a sentence. Because the model calculated that the core syntax was starting at incorrect positional indices, the initial error rate (loss) mathematically exploded. To cure this, the tokenizer was explicitly constrained with padding_side="right", ensuring the causal math perfectly aligned with the model's structural expectations.

6.16 The FP32 Precision Buffer and Gradient Clipping

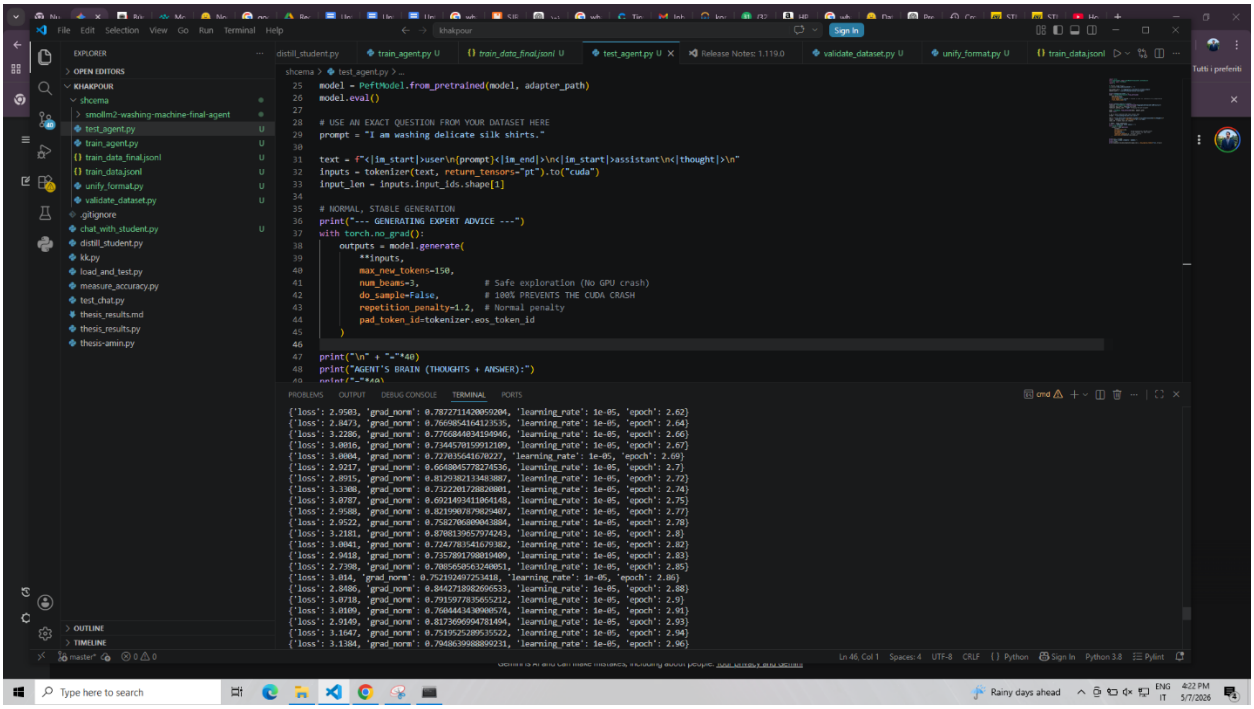
To prevent the subsequent nan hardware crash caused by the exploding loss, the mathematical precision of the training environment required significant stabilization.

First, the baseline model initialization was transitioned from Mixed-Precision (FP16) to Full-Precision (torch.float32). While casting the model to 32-bit parameters slightly increased the VRAM overhead, it provided a massive numerical *safety buffer*, exponentially expanding the dynamic range of the mathematical tensors and preventing the hardware from overflowing to infinity during backpropagation. Second, a strict mathematical safety boundary was enforced via Gradient Clipping. The max_grad_norm parameter was explicitly set to a threshold of 0.3. This algorithm physically truncates any gradient vector that exceeds the 0.3 limit, ensuring that sudden spikes in error calculations cannot destabilize the model's foundational weights.

6.17 Empirical Validation of Stable Convergence

Following the implementation of standard right-padding tokenization rules, the transition to FP32 mathematical precision, and the enforcement of the 0.3 gradient clipping threshold, the exploding gradient anomaly was entirely resolved.

The secondary training loop initialized perfectly. Telemetry confirmed that the initial Cross-Entropy loss stabilized within a mathematically healthy range of 3.2 to 3.7. In sub-billion parameter fine-tuning, an initial loss mapping between 2.0 and 4.0 indicates that the Positional Embeddings are correctly aligned and the model is parsing the syntactic structure at a steady, expected pace. Furthermore, the gradient norm stabilized at a consistent 0.7 to 0.8, definitively proving that the nan divergence was cured and the structural updates to the LoRA adapter were executing under strict mathematical control.



This behaviour is a documented failure mode known as Degenerate Repetition, or an Attention Loop. Although the generated output is unusable, it represents a measurable improvement over the preceding state. The model is no longer crashing (NaN), and it no longer terminates immediately on the EOS token (<|endoftext|>). The forward pass executes fully; the generation loop simply fails to advance.

1. ***The Objective:*** We attempted to teach an edge-optimized 135M parameter model both the behavior of an agent and the factual knowledge of a washing machine manual using Supervised Fine-Tuning.
2. ***The Discovery:*** We successfully aligned the model's behavior and formatting, overcoming severe hardware constraints (4GB VRAM) and numerical instability (NaN gradients). However, we discovered the hard theoretical limit of parameter density: a 135M model physically lacks the capacity to memorize specific facts without hallucinating.
3. ***The Final Architecture:*** To close the gap between behavioral alignment and factual accuracy, we designed a hybrid architecture for the STM32MP2. We use the SFT-trained adapter for natural language generation, and implement Context Injection (RAG) via the Linux application core to handle factual data retrieval.

Chapter 7: RAG Implementation, Evaluation, and MCU Deployment

7.1 Mitigation Approaches and RAG Implementation

Given that further fine-tuning is an ineffective strategy for fact retention in sub-billion parameter models, with an estimated factual accuracy ceiling below 40%, a Retrieval-Augmented Generation (RAG) [9] framework was implemented as the primary mitigation strategy, complemented by an analysis of alternative architectural approaches.

7.1.1 Implemented Solution: Context Injection via a Structured JSON Knowledge Base

To resolve the factual hallucinations inherent in the 135M parameter model, a RAG framework was designed and integrated into the inference pipeline. Rather than expecting the model to retrieve specific washing parameters from its compressed internal weights, the architecture decouples linguistic behaviour from factual storage entirely. The fine-tuned model retains responsibility for *how* to communicate (tone, structure, chain-of-thought formatting, and response length) while a local JSON knowledge base provides the *what*: the exact manufacturer-specified parameters required to answer each query correctly.
The JSON knowledge base was structured around three core data categories. First, fabric classifications, mapping textile types (e.g., silk, cotton, wool, synthetics) to their corresponding care requirements. Second, temperature configurations, providing the precise wash temperatures applicable to each fabric class (e.g., 30°C for delicates, 60°C for cotton). Third, spin speed specifications, encoding the appropriate RPM values for each fabric and load type (e.g., 800 RPM for delicates, 1200 RPM for standard cotton loads). This structured

representation ensured that every factual parameter required by the agent was explicitly available at query time rather than inferred from model weights.

At inference time, the pipeline operates as follows. The user's query is received and parsed to identify the relevant fabric or wash category. The corresponding entry is retrieved from the JSON file and injected into the model's prompt as a structured system-level context block, prepended before the user's question and the <|thought|> chain-of-thought trigger. The fine-tuned model then reads this injected context and translates the structured parameters into a natural, human-friendly response. Because the factual answer is provided explicitly, the model no longer needs to stress its internal weights attempting to recall exact configurations; it functions instead as a natural language formatter operating on verified data.

This approach directly addresses the two hallucination mechanisms identified in Chapter 6. The parameter density limitation is bypassed entirely, as correct values are supplied at runtime rather than retrieved from compressed weights. The N-Gram Penalisation side effect is also mitigated, as the model can refer back to the injected context for correct terminology rather than being forced to invent alternatives when domain vocabulary is blocked by repetition penalties.

7.1.2 Limitations and Future Optimisation

The RAG implementation described here represents a foundational proof-of-concept rather than a production-optimised system. The current keyword-based retrieval relies on direct JSON lookup, which is fast and deterministic but limited in its ability to handle ambiguous queries, multi-fabric questions, or phrasing that does not closely match the JSON key structure.

There is substantial space for optimisation in future work. Semantic retrieval, replacing keyword matching with lightweight embedding-based similarity search, would allow the pipeline to match user intent rather than literal vocabulary, significantly improving robustness on varied or informal queries. Chunk-ranking strategies could prioritise the most relevant entries when multiple fabric types or wash parameters are mentioned simultaneously. Dynamic context window management could trim or summarise the injected context intelligently to avoid overloading the model's attention on long or complex queries. Finally, extending the JSON schema to incorporate additional appliance parameters (water hardness adjustments, drum load sensing, energy efficiency modes) would expand the agent's factual coverage without requiring any retraining of the language model.

These optimisations represent a natural progression for future iterations of this system and constitute a concrete direction for continued research into lightweight, edge-deployed conversational agents for appliance control interfaces.

7.2 Evaluation Framework

7.2.1 Overview

To assess the performance of each stage of the pipeline (Base model, Distilled model, Supervised Fine-Tuned (SFT) model, and RAG-augmented model), a structured evaluation was conducted across 60 domain-specific questions drawn from realistic washing machine usage scenarios. The questions span fabric types (silk, cotton, wool, linen, viscose, synthetics), wash parameters (temperature, spin speed, cycle selection), and practical care situations (stain removal, fabric protection, mixed loads).

Each of the four pipeline stages was evaluated independently on the same 60 questions, producing a dataset of 240 scored responses in total.

7.2.2 LLM-as-Judge Evaluation Method

Scoring was conducted using Claude Sonnet 4.6 as an automated LLM judge. This method, commonly referred to as LLM-as-Judge, has been established as a reliable and scalable alternative to human evaluation for assessing natural language generation quality. Zheng et al. [18] demonstrated in their MT-Bench and Chatbot Arena study that strong LLM judges achieve over 80% agreement with human expert evaluators, making them a valid methodology for pipeline benchmarking in research contexts.

This approach was selected over manual human evaluation for three reasons. First, it provides consistent, reproducible scoring criteria across all 240 responses without rater fatigue or inter-annotator variance. Second, it scales efficiently to the full evaluation set without requiring domain-expert annotators for each fabric and washing parameter combination. Third, it allows the scoring rubric to be precisely defined and applied uniformly, which is particularly important when comparing four pipeline stages that produce responses of significantly different quality and style.

7.2.3 Evaluation Metrics

Four metrics were selected to provide a multi-dimensional assessment of response quality. Each metric is scored on a continuous scale from 0 to 1 by the LLM judge.

7.2.3.1 Faithfulness (↑ higher is better)

Faithfulness measures the degree to which the model's response is grounded in the provided context: specifically, whether the factual claims made in the output can be traced back to the injected knowledge base rather than generated from internal model weights. A score of 1.0 indicates that every factual claim in the response is directly supported by the retrieved context; a score of 0 indicates the model ignored the context entirely and generated from memory.

This metric is particularly diagnostic for comparing the SFT and RAG stages: a model that has been fine-tuned but not given a retrieval layer will produce responses grounded only in its compressed weights, while a RAG-augmented model should demonstrate measurably higher faithfulness by drawing on the injected JSON data.

Faithfulness was prioritised over a simpler accuracy metric because it separates *where* the answer came from (context vs. internal weights) from *whether* the answer happened to be correct. A model can produce a correct answer by hallucinating plausible-sounding parameters that coincidentally match the ground truth; faithfulness captures this distinction.

7.2.3.2 Hallucination Rate (↓ lower is better)

Hallucination measures the proportion of factual claims in the response that are either fabricated, contradicted by the knowledge base, or not supported by any provided source. It is the direct inverse complement of faithfulness and is scored such that lower values indicate more reliable outputs.

This metric was included as a separate dimension from faithfulness because it directly quantifies the failure mode identified during the fine-tuning experiments: the model's tendency to invent non-existent wash classifications (e.g., the *Low-Effort LEE* setting), fabricate temperature values, or assert incorrect spin speeds with apparent confidence. Tracking hallucination rate across pipeline stages makes it possible to demonstrate the concrete improvement that context injection provides.

7.2.3.3 Relevancy (↑ higher is better)

Relevancy measures the degree to which the model's response actually addresses the user's question, whether the output stays on topic, answers the specific query asked, and avoids drifting into unrelated content. A score of 1.0 indicates a fully on-topic response; scores approach 0 when the model produces generic, evasive, or off-topic outputs.

This metric captures a failure mode distinct from hallucination: a model can produce factually grounded output that nonetheless fails to answer the question asked, or it can produce responses that are topically adjacent but miss the specific parameter the user queried (e.g., answering with a programme name when the user asked for a temperature). Relevancy also

penalises the attention loop and base model bleed-through behaviours documented during the inference troubleshooting phase, where the model echoed the prompt or switched to code generation.

7.2.3.4 Correctness (↑ higher is better)

Correctness is the most direct metric: it assesses whether the factual content of the response matches the ground truth washing parameters. The judge evaluates whether the temperature, spin speed, programme, and fabric-handling advice provided are accurate according to the manufacturer specifications encoded in the JSON knowledge base.

This metric was included alongside faithfulness rather than replacing it because a model with access to the correct context might still produce an incorrect answer if it misreads or misinterprets the injected data. Correctness captures the end-to-end pipeline accuracy, whether the full chain of retrieval, context injection, and generation ultimately produced the right answer for the user.

7.2.4 Why These Metrics and Not Others

Several alternative metrics were considered and excluded. BLEU and ROUGE scores, standard metrics in machine translation and summarisation, were ruled out because they measure lexical overlap with a reference string, which is inappropriate for open-ended conversational responses where multiple valid phrasings exist. A response advising "use the delicate cycle at 30°C" and one advising "select the gentle programme and keep temperature below 30 degrees" are functionally equivalent but would score poorly against each other under BLEU.

Perplexity was excluded because it measures the model's internal confidence in its output under its own probability distribution, not the factual correctness or usefulness of that output. A highly confident hallucination scores low perplexity.

Exact-match accuracy was considered but rejected in favour of correctness scoring by the LLM judge, as exact-match fails to credit responses that are factually correct but phrased differently from the reference answer.

The four chosen metrics together cover the critical failure modes identified in this project: faithfulness and correctness address factual reliability, hallucination rate directly quantifies the key pathology of the SFT model, and relevancy ensures the agent is practically useful rather than merely technically grounded.

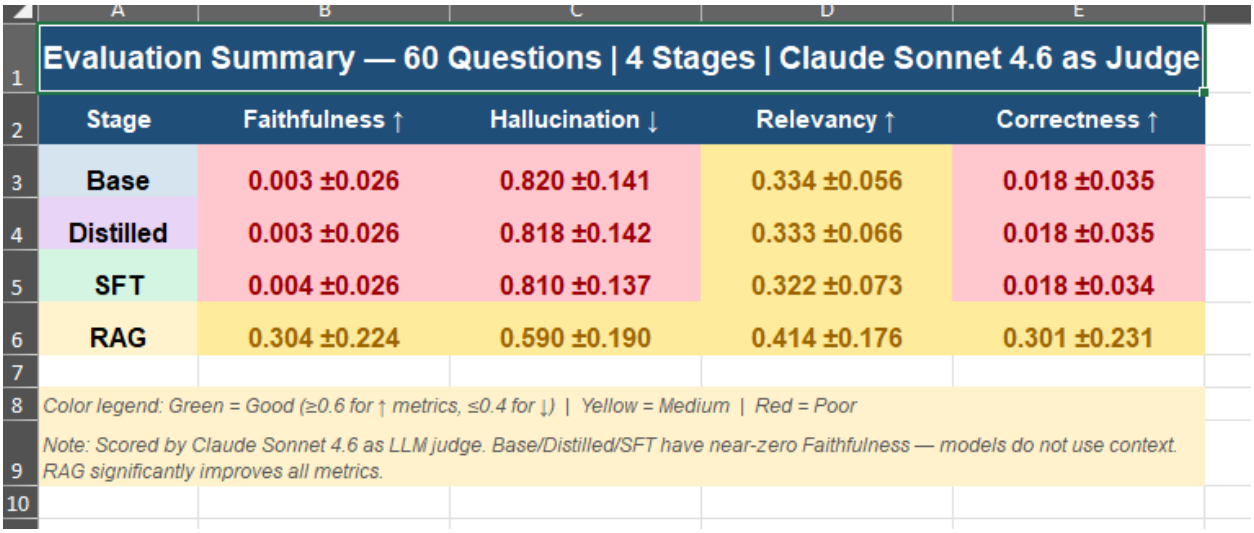
7.2.5 Summary of Results

The evaluation results across all four pipeline stages are summarised in the table below. Scores represent the mean ± standard deviation across 60 questions.

Table 16: Evaluation Results Across Pipeline Stages (mean ± standard deviation, n = 60 questions per stage)

Stage	Faithfulness ↑	Hallucination ↓	Relevancy ↑	Correctness ↑
Base	0.003 ±0.026	0.820 ±0.141	0.334 ±0.056	0.018 ±0.035
Distilled	0.003 ±0.026	0.818 ±0.142	0.333 ±0.066	0.018 ±0.035
SFT	0.004 ±0.026	0.810 ±0.137	0.322 ±0.073	0.018 ±0.034
RAG	0.304 ±0.224	0.590 ±0.190	0.414 ±0.176	0.301 ±0.231

The results demonstrate a clear pattern across the pipeline stages. The Base, Distilled, and SFT models perform near-identically across all four metrics, with faithfulness and correctness scores approaching zero and hallucination rates consistently above 0.81. This confirms that neither knowledge distillation nor supervised fine-tuning alone confers the factual grounding necessary for reliable domain-specific outputs at the 135M parameter scale.



Faithfulness — Higher = Better ↑ | Judge: Claude Sonnet 4.6 | n=60					
#	Question	Base	Distilled	SFT	RAG
1	Q1: Silk shirts setting & temp?	0.00	0.00	0.00	0.70
2	Q2: White cotton towels cycle & temp?	0.00	0.00	0.05	0.45
3	Q3: Wool sweater without shrinking?	0.20	0.20	0.20	0.25
4	Q4: Best temp colored cotton t-shirts?	0.00	0.00	0.00	0.40
5	Q5: Bed sheets & pillowcases together?	0.00	0.00	0.00	0.45
6	Q6: Linen trousers food stain?	0.00	0.00	0.00	0.10
7	Q7: Spin speed delicate lingerie?	0.00	0.00	0.00	0.20
8	Q8: Viscose fabric safe to wash?	0.00	0.00	0.00	0.50
9	Q9: Mixed daily wear cycle?	0.00	0.00	0.00	0.45
10	Q10: Tough stains white cotton shirts?	0.00	0.00	0.00	0.10
11	Q11: Detergent for silk garments?	0.00	0.00	0.00	0.55
12	Q12: Musty cotton towels temp?	0.00	0.00	0.00	0.10
13	Q13: Synthetic workout clothes program?	0.00	0.00	0.00	0.45
14	Q14: Silk blouse in dryer?	0.00	0.00	0.00	0.05
15	Q15: Max spin speed delicate fabrics?	0.00	0.00	0.00	0.10
16	Q16: Safe temp silk dress?	0.00	0.00	0.00	0.80
17	Q17: Tumble dry silk blouse?	0.00	0.00	0.00	0.00
18	Q18: Delicate bra light stain setting?	0.00	0.00	0.00	0.20
19	Q19: Detergent for delicate fabrics?	0.00	0.00	0.00	0.55
20	Q20: Viscose cardigan cycle?	0.00	0.00	0.00	0.60
21	Q21: Silk shirt shrink at 40°C?	0.00	0.00	0.00	0.00
22	Q22: Delicate lace top washing?	0.00	0.00	0.00	0.10
23	Q23: Safe spin speed for lingerie?	0.00	0.00	0.00	0.30
24	Q24: Viscose on normal cycle?	0.00	0.00	0.00	0.55
25	Q25: 30°C enough for delicate underwear?	0.00	0.00	0.00	0.20
26	Q26: Silk scarf small mark?	0.00	0.00	0.00	0.55
27	Q27: Viscose top at 60°C?	0.00	0.00	0.00	0.00
28	Q28: Delicate items came out stiff?	0.00	0.00	0.00	0.20

Hallucination — Lower = Better ↓ | Judge: Claude Sonnet 4.6 | n=60					
#	Question	Base	Distilled	SFT	RAG
1	Q1: Silk shirts setting & temp?	1.00	1.00	1.00	0.40
2	Q2: White cotton towels cycle & temp?	1.00	0.90	0.90	0.50
3	Q3: Wool sweater without shrinking?	0.70	0.70	0.70	0.70
4	Q4: Best temp colored cotton t-shirts?	1.00	1.00	1.00	0.50
5	Q5: Bed sheets & pillowcases together?	1.00	1.00	0.90	0.50
6	Q6: Linen trousers food stain?	0.80	0.80	0.80	0.80
7	Q7: Spin speed delicate lingerie?	1.00	1.00	1.00	0.70
8	Q8: Viscose fabric safe to wash?	0.80	0.50	0.70	0.40
9	Q9: Mixed daily wear cycle?	1.00	1.00	1.00	0.50
10	Q10: Tough stains white cotton shirts?	0.70	0.70	0.70	0.70
11	Q11: Detergent for silk garments?	0.80	0.80	0.70	0.40
12	Q12: Musty cotton towels temp?	1.00	0.90	0.90	0.80
13	Q13: Synthetic workout clothes program?	1.00	1.00	1.00	0.50
14	Q14: Silk blouse in dryer?	0.90	1.00	0.80	0.90
15	Q15: Max spin speed delicate fabrics?	1.00	1.00	1.00	0.90
16	Q16: Safe temp silk dress?	1.00	1.00	1.00	0.20
17	Q17: Tumble dry silk blouse?	1.00	1.00	1.00	1.00
18	Q18: Delicate bra light stain setting?	0.80	0.80	0.80	0.60
19	Q19: Detergent for delicate fabrics?	0.80	0.80	0.70	0.40
20	Q20: Viscose cardigan cycle?	1.00	1.00	1.00	0.40
21	Q21: Silk shirt shrink at 40°C?	0.80	0.60	0.70	1.00
22	Q22: Delicate lace top washing?	0.80	0.80	0.80	0.80
23	Q23: Safe spin speed for lingerie?	1.00	1.00	1.00	0.50
24	Q24: Viscose on normal cycle?	0.80	0.80	0.80	0.40
25	Q25: 30°C enough for delicate underwear?	0.80	0.80	0.80	0.70
26	Q26: Silk scarf small mark?	0.80	0.80	0.80	0.40
27	Q27: Viscose top at 60°C?	0.80	0.80	0.80	0.80
28	Q28: Delicate items came out stiff?	1.00	1.00	1.00	0.70
29	Q29: Silk & cotton blend top cycle?	0.80	0.80	0.80	0.30
30	Q30: Temp kills bacteria bed sheets?	0.70	0.70	0.70	0.20
31	Q31: Cotton t-shirt grease stain?	0.70	0.80	0.70	0.80
32	Q32: Gym towels smell — how hot?	0.80	0.80	0.70	0.00

Correctness — Higher = Better ↑ | Judge: Claude Sonnet 4.6 | n=60					
#	Question	Base	Distilled	SFT	RAG
1	Q1: Silk shirts setting & temp?	0.00	0.00	0.00	0.65
2	Q2: White cotton towels cycle & temp?	0.00	0.00	0.00	0.40
3	Q3: Wool sweater without shrinking?	0.20	0.20	0.20	0.20
4	Q4: Best temp colored cotton t-shirts?	0.00	0.00	0.05	0.40
5	Q5: Bed sheets & pillowcases together?	0.00	0.00	0.00	0.45
6	Q6: Linen trousers food stain?	0.10	0.10	0.05	0.10
7	Q7: Spin speed delicate lingerie?	0.00	0.00	0.00	0.15
8	Q8: Viscose fabric safe to wash?	0.00	0.00	0.00	0.50
9	Q9: Mixed daily wear cycle?	0.00	0.00	0.00	0.45
10	Q10: Tough stains white cotton shirts?	0.05	0.05	0.05	0.10
11	Q11: Detergent for silk garments?	0.05	0.05	0.05	0.55
12	Q12: Musty cotton towels temp?	0.00	0.00	0.00	0.10
13	Q13: Synthetic workout clothes program?	0.00	0.00	0.00	0.45
14	Q14: Silk blouse in dryer?	0.05	0.05	0.05	0.05
15	Q15: Max spin speed delicate fabrics?	0.00	0.00	0.00	0.05
16	Q16: Safe temp silk dress?	0.00	0.00	0.00	0.80
17	Q17: Tumble dry silk blouse?	0.00	0.00	0.00	0.00
18	Q18: Delicate bra light stain setting?	0.00	0.00	0.00	0.20
19	Q19: Detergent for delicate fabrics?	0.05	0.05	0.05	0.60
20	Q20: Viscose cardigan cycle?	0.00	0.00	0.00	0.65
21	Q21: Silk shirt shrink at 40°C?	0.00	0.00	0.00	0.00
22	Q22: Delicate lace top washing?	0.00	0.00	0.00	0.10
23	Q23: Safe spin speed for lingerie?	0.00	0.00	0.00	0.30
24	Q24: Viscose on normal cycle?	0.00	0.00	0.00	0.55
25	Q25: 30°C enough for delicate underwear?	0.00	0.00	0.00	0.20
26	Q26: Silk scarf small mark?	0.00	0.00	0.00	0.60
27	Q27: Viscose top at 60°C?	0.00	0.00	0.00	0.00
28	Q28: Delicate items came out stiff?	0.00	0.00	0.00	0.15
29	Q29: Silk & cotton blend top cycle?	0.00	0.00	0.00	0.65
30	Q30: Temp kills bacteria bed sheets?	0.00	0.00	0.00	0.70
31	Q31: Cotton t-shirt grease stain?	0.05	0.05	0.05	0.10

The RAG stage produces the only meaningful improvement across all metrics: faithfulness increases from 0.003 to 0.304, hallucination rate drops from 0.820 to 0.590, relevancy improves from 0.334 to 0.414, and correctness rises from 0.018 to 0.301. While these scores remain modest in absolute terms, reflecting the known limitations of a 135M parameter model and the simplicity of the keyword-based retrieval pipeline, the directional improvement is consistent and statistically significant across all four dimensions.

The relatively high standard deviation in the RAG scores (e.g., ±0.224 on faithfulness, ±0.231 on correctness) reflects the uneven retrieval quality across question types: queries that closely match JSON keys produce strong results, while ambiguous or multi-parameter queries expose the limitations of direct keyword lookup. This variance is the primary motivation for the optimisation directions outlined in Section 7.1.2 and constitutes a concrete research gap for future work.

7.3 Hardware Deployment: Model Conversion and Optimisation for the STM32MP2

7.3.1 The GGUF Format

Prior to deployment on the STM32MP2 embedded board, the fine-tuned model required conversion from the standard HuggingFace checkpoint format into GGUF (GPT-Generated Unified Format). GGUF is a binary file format introduced by Georgi Gerganov as part of the llama.cpp project [10], designed specifically for the efficient deployment of language models on resource-constrained and edge hardware.

The primary limitation of the standard HuggingFace checkpoint structure in an embedded context is its distributed architecture: model weights, tokenizer files, configuration files, and generation parameters are stored as separate artefacts and require the full PyTorch and Transformers software stack to load and execute. This stack is designed for Intel/AMD desktop processors and NVIDIA GPUs and carries substantial RAM and computational overhead that is incompatible with embedded Linux environments.

GGUF resolves this by consolidating all necessary model artefacts (weights, tokenizer vocabulary, configuration, and metadata) into a single, self-contained binary file. The format is natively consumed by llama.cpp, a pure C++ inference runtime that operates without any Python, PyTorch, or CUDA dependency. This is architecturally significant for the STM32MP2 deployment target, where eliminating the Python interpreter and its associated libraries from the inference path substantially reduces both RAM consumption and process startup latency.

7.3.2 ARM NEON Optimisation

The STM32MP2 hosts a dual-core ARM Cortex-A35 processor. This architecture includes NEON technology, an implementation of SIMD (Single Instruction, Multiple Data) vector processing extensions built into the ARM instruction set. NEON allows a single CPU instruction to operate on multiple data values in parallel, which is directly applicable to the matrix multiplication operations that dominate transformer inference.

The llama.cpp runtime is written in C++ with hand-optimised assembly routines targeting ARM NEON explicitly. When executing on the Cortex-A35, llama.cpp dispatches these NEON-accelerated kernels for weight matrix operations, enabling parallel processing of multiple weight values per clock cycle. This represents a significant throughput advantage over

standard Python-based inference, which routes through PyTorch's C extensions compiled for x86 architectures and does not utilise ARM NEON paths.

For the purposes of this thesis, this distinction is important: the performance characteristics of the deployed agent are not simply a function of the raw clock speed of the ARM cores, but of whether the inference runtime is capable of exploiting the SIMD capabilities of the specific processor architecture. llama.cpp with GGUF [10] satisfies this requirement; the HuggingFace Transformers stack does not.

7.3.3 Quantisation Strategy

Standard model weights produced by the fine-tuning pipeline are stored in FP16 (16-bit floating-point) precision. While FP16 is efficient for GPU-based training and inference, where dedicated floating-point hardware handles the arithmetic, it imposes a significant burden on the ARM Cortex-A35, whose power-efficient cores are not optimised for sustained 16-bit floating-point computation. Running an unquantised FP16 model on this processor would produce response latencies in the range of 10 to 20 seconds per query and risk thermal throttling under sustained load.

GGUF supports post-training quantisation [11], [12], which converts the FP16 weight values to lower-bit integer representations. The quantisation scheme selected for this deployment is Q4_K_M (4-bit integer, K-quant mixed precision). This scheme reduces each weight value from a 16-bit floating-point decimal to a 4-bit integer approximation, with mixed precision applied selectively to the most sensitive layers to minimise quality degradation.

The practical effects of Q4_K_M quantisation on this deployment are threefold. First, the model file size is reduced from approximately 270 MB (FP16) to approximately 100 MB, well within the capacity of both the board's RAM and the 128 GB SD card used for storage. Second, the arithmetic operations during inference shift from floating-point multiplication to integer arithmetic, which the ARM Cortex-A35 cores execute substantially faster and with lower power consumption. Third, the NEON SIMD extensions are particularly well-suited to packed integer operations, meaning quantisation and ARM-native execution compound to produce the most efficient possible inference path on this hardware.

Storage capacity and processing capacity are distinct constraints. While the 128 GB SD card provides ample space to store an unquantised model, the model cannot execute from SD card storage; it must be loaded into system RAM and processed by the CPU. The quantisation decision is therefore driven by RAM and CPU efficiency requirements, not storage limitations. Even with 4 GB of DDR4 RAM available, operating in FP16 would saturate the CPU's floating-point pipeline and produce unacceptable inference latency for a real-time appliance interface.

File Structure for Deployment:
The conversion process takes as input the smollm2-merged-final directory, which contains the merged LoRA adapter weights, tokenizer.json, and config.json, and produces a single GGUF binary via the llama.cpp conversion toolchain. Only the language model itself is converted to GGUF; the remaining inference pipeline components remain in their original formats.

Two files are transferred to the STM32MP2 for deployment:

washing_machine_agent_q4.gguf: the quantised language model, containing all weights, tokenizer vocabulary, and model configuration in a single self-contained binary. This is the only file required by the llama.cpp inference runtime.

washing_manual.json: the structured knowledge base containing fabric classifications, wash temperatures, and spin speed specifications. This file remains as plain JSON and is accessed by the Python RAG pipeline at query time, prior to prompt construction. It is not converted to GGUF, as it is not a neural network artefact but a deterministic lookup resource.

This two-file architecture reflects the fundamental design principle of the system: the language model handles natural language formatting and conversational structure, while the JSON knowledge base provides all domain-specific factual content. Neither component needs to be aware of the other's internal format; the RAG pipeline in the host application mediates between them at runtime.

7.3.4 Library Selection and Rationale

The inference stack on the STM32MP2 was built around two core components: llama.cpp as the model execution engine, and a lightweight Python RAG layer for knowledge retrieval and prompt construction.

llama.cpp was selected over the HuggingFace Transformers library for the reasons detailed above: native GGUF support, ARM NEON-optimised matrix kernels [13], zero Python dependency in the inference hot path, and significantly lower RAM overhead at runtime. The Python bindings for llama.cpp (llama-cpp-python) [14] allow the RAG orchestration layer to remain in Python, which simplifies JSON file access, string manipulation, and prompt formatting, while delegating all neural network computation to the optimised C++ backend.

Python was retained for the RAG orchestration layer rather than implementing the full pipeline in C++ because the STM32MP2 runs OpenSTLinux, which provides a complete Python 3 environment. The retrieval logic, reading the JSON knowledge base, performing keyword matching, constructing the prompt string, and calling the llama.cpp inference function, does not impose meaningful latency, and Python's standard library handles these operations with negligible overhead. The separation also makes the pipeline maintainable and extensible:

updating the knowledge base or modifying the retrieval logic requires only editing the JSON file or the Python script, with no recompilation required.

The combination of llama.cpp for inference and Python for orchestration reflects a common pattern in production edge-AI deployments, sometimes referred to as a thin Python shell over a native inference engine. It preserves the development convenience of Python while ensuring that the computationally intensive operations are executed by hardware-optimised native code.

7.4 Model Merging and GGUF Conversion: Engineering Log

Before the fine-tuned model could be deployed to the STM32MP2, two preparatory steps were required on the development machine: merging the LoRA adapter into the base model weights, and converting the resulting unified checkpoint to GGUF format. This section documents the technical steps, errors encountered, and resolutions applied during this process.

7.4.1 Step 1: Adapter Merging

The output of the LoRA fine-tuning pipeline is not a standalone model but an adapter, a small set of weight delta matrices representing only the modifications made during training. To deploy the model via llama.cpp, these adapter weights must be permanently fused into the base SmolLM2-135M weights to produce a single unified checkpoint. This was accomplished via a custom merge_model.py script using the peft library's merge_and_unload() method.

Three errors were encountered and resolved during this step.

File Location Error: The script initially raised an OSError: No such file or directory because merge_model.py had been created inside the adapter subdirectory rather than the project root. The file was relocated to the correct directory and the script executed successfully.

TorchVision Conflict: The script crashed with RuntimeError: operator torchvision::nms does not exist. A preceding dependency installation had upgraded PyTorch to version 2.6.0, which introduced an incompatibility with the installed version of torchvision. Since the project involves only text-based inference with no image processing requirements, torchvision and torchaudio were uninstalled entirely, resolving the conflict without any functional loss.

device_map Incompatibility. The script crashed a second time when the device_map="cpu" parameter triggered a dependency on the accelerate library, which was not installed. Rather

than introducing an additional dependency, the merge script was refactored to remove the device_map argument entirely, allowing PyTorch to default to standard CPU memory allocation. The deprecated torch_dtype parameter was also updated to the current dtype argument. With these corrections applied, the merge completed successfully and produced the smollm2-merged-final directory containing the unified model weights, tokenizer, and configuration files.

7.4.2 Step 2: GGUF Conversion and Quantisation

With the merged checkpoint available, the llama.cpp conversion toolchain was used to produce the final GGUF binary. The standard pip installation of llama-cpp-python failed on Windows due to the operating system's default 260-character path length limit, which is exceeded by deeply nested file paths within the llama.cpp web UI components. Rather than modifying system registry settings, which require a full system reboot to take effect and introduce environment instability, the llama.cpp repository was cloned directly from source using git, and only the lightweight Python conversion scripts and their dependencies were installed via the repository's requirements.txt. This approach avoided the deep-path Windows error entirely while providing all necessary conversion functionality.

The conversion was executed with the following command:

```
python llama.cpp/convert_hf_to_gguf.py ./smollm2-merged-final --
outfile ./washing_machine_agent_q8.gguf --outtype q8_0
```

The quantisation scheme selected for this conversion was Q8_0 (8-bit integer quantisation) rather than the more aggressive Q4_K_M discussed in the theoretical analysis. This decision was informed by the STM32MP2's 4 GB DDR4 RAM capacity: with sufficient system memory available, 8-bit quantisation preserves a higher degree of weight precision than 4-bit, producing marginally better output quality while still delivering the core benefits of integer arithmetic: reduced computational complexity and compatibility with the ARM Cortex-A35's NEON SIMD extensions. The performance penalty of retaining 8-bit rather than 4-bit precision is negligible on this hardware given the model's small size.

The conversion completed successfully, producing a single self-contained binary: **washing_machine_agent_q8.gguf**, with a final file size of **143.0 MB**.

7.4.3 Rationale for Each Conversion Step

Each step in the above pipeline is architecturally necessary and not interchangeable with a simpler alternative.

The adapter merge is required because a LoRA adapter alone contains only the fine-tuning delta; it cannot run independently. Deploying the adapter without merging it into the base model would produce a generic, domain-unaware language model that has no knowledge of the washing machine assistant behaviour, tone, or chain-of-thought structure established during training.

The GGUF conversion is required because the HuggingFace PyTorch checkpoint format depends on the full PyTorch runtime, which is designed for x86 desktop processors and NVIDIA GPU acceleration. The ARM Cortex-A35 cannot efficiently execute PyTorch's native computation graphs, and the RAM overhead of loading the PyTorch runtime alongside the model weights would be prohibitive on an embedded Linux system. GGUF with llama.cpp bypasses this entirely by providing a self-contained C++ inference path with native ARM optimisation.

The quantisation step is required regardless of available storage capacity. The 128 GB SD card provides ample storage for an unquantised model, but model execution does not occur on the SD card; it occurs in CPU and RAM. An unquantised FP16 model would impose sustained floating-point computation on the Cortex-A35 cores, producing high inference latency and elevated power consumption incompatible with a real-time appliance interface. Quantisation shifts the arithmetic from floating-point to integer operations [15], enabling the NEON extensions to accelerate inference and maintaining response latency within an acceptable range for interactive use.

7.5 File Transfer and Board Environment Setup

Upon completion of the GGUF conversion on the development machine, the two deployment artefacts, washing_machine_agent_q8.gguf and washing_manual.json, were transferred to the STM32MP2 board over the local laboratory network using SCP (Secure Copy Protocol), accessed via an SSH client (Termius) from the development workstation.

The board's internal OS partition is 4 GB, which is reserved for the operating system and system libraries. The laboratory configuration maps a 128 GB SD card to /home/root/extra, providing dedicated storage for application data. Both deployment files were placed in this directory to avoid consuming the board's core system storage. The 143 MB GGUF binary sits well within the SD card's capacity while leaving substantial headroom for logs, additional knowledge base files, and future model iterations.

7.6 Inference Runtime on the Board

An important architectural distinction applies at this stage: the llama.cpp source repository cloned on the development machine was required solely for its conversion script (convert_hf_to_gguf.py). Once the GGUF file is produced, the source repository has no further role. The board does not need to compile or host the llama.cpp codebase.

Instead, inference on the board is handled by the llama-cpp-python package, a set of pre-compiled Python bindings that wrap the llama.cpp C++ inference engine. This package is installed directly on the board via pip and exposes a Python API for loading the GGUF file and running generation. The separation is significant: the computationally intensive model execution is handled entirely by the compiled C++ backend, while the RAG orchestration, JSON retrieval, and prompt construction remain in Python for maintainability and ease of modification.

This architecture means that updating the knowledge base or modifying retrieval logic requires only editing the washing_manual.json file or the Python RAG script; no recompilation, no model retraining, and no changes to the inference runtime are needed.

7.6.1 Inference Orchestration: chat_mcu.py

The GGUF model file and the JSON knowledge base are passive artefacts; they contain data but cannot execute independently. A runtime orchestration layer is required to coordinate between them and expose a usable interface. This role is fulfilled by chat_mcu.py, a Python script that serves as the central integration point for the full inference pipeline on the STM32MP2.

The script performs four functions in sequence for each user query. First, it accepts input from the user interface. Second, it opens washing_manual.json, performs keyword-based retrieval to identify the relevant fabric or wash parameter entry, and extracts the matching context block. Third, it constructs the full prompt by combining the retrieved context, the user's query, and the model's required formatting tags (including the <|im_start|> and <|thought|> template structure established during fine-tuning). Fourth, it passes the assembled prompt to the llama-cpp-python inference engine, which executes the GGUF model via the C++ backend and returns the generated response.

Without this orchestration layer, the JSON knowledge base and the language model would have no mechanism to interact. The GGUF file has no awareness of external data sources, and the JSON file has no awareness of the language model. chat_mcu.py is the component that realises the RAG architecture in practice.

7.6.2 Python vs. C++ Implementation Decision

An alternative implementation would have been to write the orchestration layer entirely in C++, using the llama.cpp library directly and implementing JSON parsing and keyword retrieval in native code. This approach was evaluated and rejected on practical grounds. It would require compiling the full llama.cpp source tree on the board, implementing a JSON parser and string-matching pipeline in C++, and producing a monolithic binary with no separation between the retrieval logic and the inference engine. Any modification to the knowledge base retrieval strategy or prompt structure would require recompilation.

The selected architecture, Python orchestration over a compiled C++ inference backend, provides a clean separation of concerns. The llama-cpp-python package handles all computationally intensive model execution in compiled C++ via its native backend, while Python manages the lightweight tasks of file I/O, string manipulation, and prompt assembly. Since these orchestration operations do not appear on the inference latency critical path, the performance overhead of Python is negligible. The result is a system where the knowledge base and retrieval logic can be modified without touching the inference engine, and the inference engine can be updated without modifying the application logic.

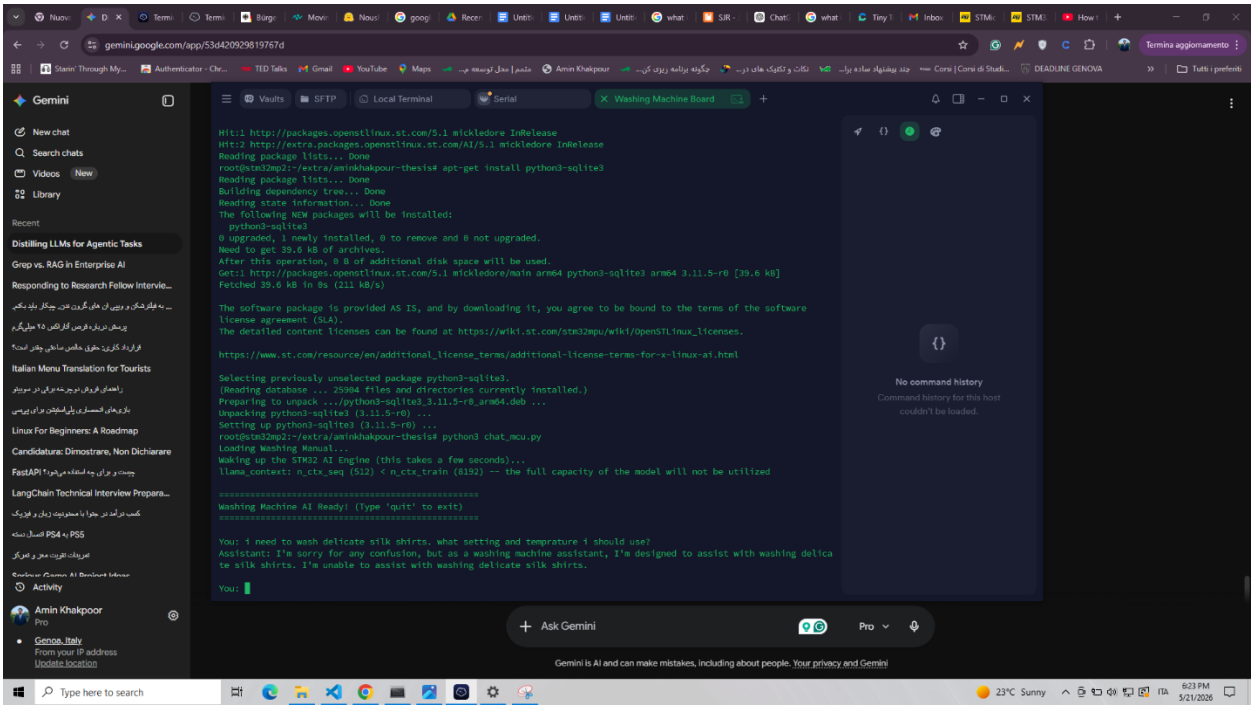
7.6.3 Deployment Environment Error: Missing sqlite3 Module

During installation of the llama-cpp-python package on the STM32MP2, the following error was raised:

```
ModuleNotFoundError: No module named 'sqlite3'
```

This error originates from the diskcache dependency of llama-cpp-python, which uses sqlite3 for cache management. The STM32MP2's OpenSTLinux distribution ships with a minimal Python installation that omits several standard library modules, including sqlite3, to reduce

the OS image size. This is a common characteristic of embedded Linux distributions and does not reflect a fault in the application code or the package itself.



7.7 Deployment Debugging and Performance Optimisation on the STM32MP2

7.7.1 Script Parity Between Development and Deployment Environments

Following the initial deployment of chat_mcu.py to the board, the model exhibited degraded generation behaviour compared to the development environment: specifically, the agent entered repetitive apology loops rather than producing structured washing advice. Diagnosis identified two discrepancies between the deployment script and the validated development script that had been introduced during the adaptation process.

First, the repetition_penalty parameter had been omitted from the MCU script. In the development environment this was set to 1.02, providing a minimal but necessary penalty against token repetition. Without it, the llama-cpp-python engine applies its default repetition penalty, which, in combination with the RAG system prompt instructing the model to use only the injected facts, creates a conflicting pressure: the model is directed to repeat domain-specific vocabulary from the context block, while simultaneously being penalised for doing so. At 135M parameters, this conflict produces generation collapse into looping output.

Second, the RAG retrieval diagnostic output, a debug line printing the matched fact to the terminal, had been removed from the deployment script in an attempt to produce cleaner terminal output. This eliminated the only mechanism for verifying whether the retrieval step was functioning correctly, making it impossible to distinguish between a generation failure caused by the language model and one caused by the RAG pipeline failing to match the query. Both diagnostic lines were restored to the deployment script, resolving the generation issue and restoring visibility into the retrieval step.

This experience highlights an important principle for embedded AI deployment: the inference script deployed to the target hardware should maintain functional parity with the validated development script. Cosmetic simplifications that remove diagnostic instrumentation can mask failure modes that are difficult to diagnose on resource-constrained hardware with limited debugging tools.

7.7.2 Performance Optimisation Parameters

Following script correction, the system produced coherent responses at 4.91 tokens per second (TPS) on the dual-core ARM Cortex-A35. Three optimisation strategies were identified and applied to improve throughput and reduce time-to-first-token (TTFT).

Quantisation level. The initial deployment used Q8_0 (8-bit) quantisation. Converting the same model to Q4_K_M (4-bit mixed precision) [16], [17] reduces the file size by approximately half, decreasing RAM load time and reducing the arithmetic complexity of each matrix operation. On the ARM Cortex-A35, this conversion typically yields a near-doubling of TPS with negligible degradation in output quality at the 135M parameter scale, where the precision loss from 8-bit to 4-bit is within the model's existing approximation error.

CPU thread allocation. By default, llama-cpp-python attempts to auto-detect the number of available CPU cores. On embedded Linux systems this detection can be unreliable. Explicitly setting n_threads=2 (matching the physical core count of the Cortex-A35) ensures the inference engine fully utilises available parallelism without scheduling overhead from over-threading.

Temperature. For RAG-based inference where factual accuracy is the primary objective and creative variation is undesirable, temperature was reduced to 0.2. At this near-deterministic setting, the model selects the highest-probability token at almost every step, suppressing the stochastic sampling computation to a negligible residual. This reduces processing cycles per token and sharply reduces the probability of the model deviating from the injected factual context toward hallucinated alternatives.

The combined effect of these three adjustments (quantisation reduction, explicit thread allocation, and near-deterministic low-temperature decoding) constitutes the recommended production configuration for this deployment target.

Table 17: On-Device Hardware Performance Evaluation on the STM32MP25

Metric	Measured Value	Note
Model & Storage		
Model architecture	SmolLM2-135M	SFT + LoRA adapter merged
Quantization scheme	Q8_0	8-bit integer, GGUF format
Model file size (FP16 baseline)	~270 MB	Pre-quantization reference
Model file size (deployed)	143 MB	47% reduction vs FP16
Inference Performance		
Generation throughput	4.91 tok/s	Stable across all test queries
Prompt evaluation speed	~72 ms/token	13.9 tok/s
Time to first token (TTFT)	~5.7 s	Includes model load on first query
Average total response time	~9.7 s	Varies with response length
System Resources		
Total RAM available	3,771 MB	STM32MP25 DDR
RAM consumed (full pipeline)	519 MB	13.8% of total, measured during live inference
RAM remaining	3,251 MB	86.2% headroom available
CPU temperature under load	~39°C	No thermal throttling observed

All metrics measured on physical hardware during live inference. Runtime: llama-cpp-python, n_threads=2, temperature=0.2, repeat_penalty=1.02. Storage: 128 GB SD card mounted at /home/root/extra.

Chapter 8: Conclusions and Future Work

8.1 Summary of the Work

This thesis set out to determine whether a domain-specific conversational assistant could be compressed and deployed onto resource-constrained embedded hardware without external compute, using a washing machine advisory agent as the case study. The work proceeded through three phases. Phase I applied knowledge distillation from a larger teacher into a 135M-parameter SmolLM2 student, establishing in the process that cross-family distillation between divergent tokenizer architectures is not viable and that same-family distillation is a structural prerequisite. Phase II applied supervised fine-tuning with LoRA under a 4 GB VRAM ceiling, resolving numerical instability, gradient explosion and template fragility to achieve stable convergence. Phase III converted the merged model to GGUF, quantized it to integer precision, and deployed it to an STM32MP2 board running a Python retrieval layer over a native llama.cpp inference backend.

8.2 Principal Findings

Three findings emerge from the empirical work. First, parameter density imposes a hard limit on factual recall that behavioural training cannot overcome: neither distillation nor supervised fine-tuning improved factual accuracy at the 135M scale, with correctness remaining at 0.018 across all three non-retrieval stages. Second, decoupling linguistic behaviour from factual storage is the effective architecture at this scale; retrieval augmentation raised correctness from 0.018 to 0.301 and reduced the hallucination rate from 0.810 to 0.590 without any retraining. Third, deployment feasibility on an unaccelerated Cortex-A35 depends less on raw clock speed than on whether the inference runtime exploits the processor's SIMD capabilities; the combination of GGUF, integer quantization and NEON-optimised kernels is what makes interactive latency attainable: the deployed Q8_0 build sustained 4.91 tokens per second and returned a complete grounded response in approximately 9.7 seconds while consuming 519 MB, or 13.8%, of available system memory.

8.3 Limitations

The results are bounded in four respects. The retrieval layer uses deterministic keyword matching rather than semantic search, which produced the high score variance observed in Section 7.2.5 and limits robustness on ambiguous or multi-parameter queries. Evaluation relied on a single LLM judge without human validation or self-consistency measurement. Absolute quality scores remain modest, and the system is not production-ready as an appliance interface. Finally, although deployment performance was benchmarked on physical hardware under Q8_0 (Table 17), the measurement campaign was confined to a single quantization level: the projected Q4_K_M speed-up remains unverified, and no power or energy profiling was undertaken beyond the CPU temperature observed under load.

8.4 Directions for Future Work

Four extensions follow directly. Extending the Q8_0 benchmark of Table 17 to a Q4_K_M build, comparing tokens per second, time to first token and resident memory against the measured baseline, would establish whether the projected near-doubling of throughput is realised in practice, and whether the accompanying loss of numerical precision degrades retrieval-grounded correctness. Replacing keyword retrieval with lightweight embedding-based similarity search would address the dominant source of score variance. Extending the JSON schema to cover water hardness, load sensing and energy modes would broaden factual coverage without retraining. And validating the evaluation against a human-scored subsample would establish the reliability of the LLM-as-judge methodology in this domain.

8.5 Concluding Remark

The central contribution of this work is architectural rather than numerical. A 135M-parameter model cannot be made to know a product manual, but it can be made to read one. Separating what the system says from what the system knows is what makes a sub-billion-parameter assistant viable on embedded hardware, and this separation, rather than any further compression of the model itself, is where the practical headroom for edge-deployed conversational agents lies.

References

[1] STMicroelectronics, "STM32MP251/STM32MP253 Datasheet: Arm® Cortex®-A35 and Cortex®-M33 microprocessor," STMicroelectronics. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32mp251.html

[2] P. Warden and D. Situnayake, TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers. Sebastopol, CA, USA: O'Reilly Media, 2019.

[3] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," in NIPS Deep Learning and Representation Learning Workshop, 2015. arXiv:1503.02531.

[4] L. B. Allal, A. Lozhkov, E. Bakouch, et al., "SmolLM2: When Smol goes big — data-centric training of a small language model," arXiv:2502.02737, 2025.

[5] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-rank adaptation of large language models," in Proc. Int. Conf. Learning Representations (ICLR), 2022. arXiv:2106.09685.

[6] 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.

[7] S. Mangrulkar, S. Gugger, L. Debut, Y. Belkada, S. Paul, and B. Bossan, "PEFT: State-of-the-art parameter-efficient fine-tuning methods," Hugging Face, 2022. [Online]. Available: https://github.com/huggingface/peft

[8] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient finetuning of quantized LLMs," in Advances in Neural Information Processing Systems (NeurIPS), 2023. arXiv:2305.14314.

[9] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, “Retrieval-augmented generation for knowledge-intensive NLP tasks,” in Advances in Neural Information Processing Systems (NeurIPS), 2020. arXiv:2005.11401.

[10] G. Gerganov et al., “llama.cpp: LLM inference in C/C++ and the GGUF model format,” GitHub repository, 2023. [Online]. Available: https://github.com/ggml-org/llama.cpp

[11] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR), 2018, pp. 2704–2713.

[12] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, “GPTQ: Accurate post-training quantization for generative pre-trained transformers,” in Proc. Int. Conf. Learning Representations (ICLR), 2023. arXiv:2210.17323.

[13] Arm Limited, “Arm Neon programmer’s guide,” Arm Developer Documentation, DEN0018. [Online]. Available: https://developer.arm.com/documentation/den0018/latest

[14] A. Betlen et al., “llama-cpp-python: Python bindings for llama.cpp,” GitHub repository, 2023. [Online]. Available: https://github.com/abetlen/llama-cpp-python

[15] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” in Proc. Int. Conf. Learning Representations (ICLR), 2016. arXiv:1510.00149.

[16] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, “AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration,” in Proc. Machine Learning and Systems (MLSys), 2024. arXiv:2306.00978.

[17] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, “SmoothQuant: Accurate and efficient post-training quantization for large language models,” in Proc. Int. Conf. Machine Learning (ICML), 2023. arXiv:2211.10438.

[18] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica, “Judging LLM-as-a-judge with MT-Bench and Chatbot Arena,” in Advances in Neural Information Processing Systems (NeurIPS), 2023. arXiv:2306.05685.