TECHNICAL WHITE PAPER · QUANTFORGE

QuantForge: A Native Apple Silicon Platform for Large Language Model Quantization

A native SwiftUI platform for quantizing, benchmarking, and managing large language models on Apple Silicon.

GGUF MLX SwiftUI iPad / Mac On-Device

ABSTRACT

QuantForge is a native macOS and iPadOS application that brings large language model quantization — the process of reducing weight precision to shrink model size and accelerate inference — out of terminal scripts and into a direct-manipulation interface. It wraps two established quantization backends, `llama-quantize` (GGUF) and `mlx_lm.convert` (MLX), behind a single SwiftUI workflow: import a model, inspect its architecture, choose a target format, and receive live progress alongside before/after benchmarks. This paper examines why quantization has remained a command-line-only discipline despite its centrality to practical on-device LLM deployment, describes QuantForge's dual-pipeline architecture and its metadata-driven approach to model inspection, and argues that making the size/speed/quality tradeoff visible and interactive — rather than a one-shot terminal invocation — changes how practitioners choose a quantization format.

01 Introduction

Every large language model shipped for local inference passes through a quantization decision. A 7-billion-parameter model at 16-bit precision occupies roughly 14 GB — too large to load comfortably on most consumer hardware, and far larger than necessary for the effective precision most inference workloads require. Quantization maps these 16-bit weights to lower-precision representations (8-bit, 4-bit, or mixed schemes), trading a small, usually acceptable increase in perplexity for large reductions in file size, memory footprint, and — because memory bandwidth rather than compute is the bottleneck in autoregressive decoding — inference latency [1].

The tooling for this decision has not kept pace with its importance. `llama.cpp`'s `llama-quantize` and `mlx-lm`'s `mlx_lm.convert` are both mature, well-engineered command-line tools, but they are tools: a practitioner must already know that GGUF and MLX are different ecosystems with different quantization vocabularies, must locate or build the correct binary, must construct the correct invocation from memory or documentation, and must separately write or find a benchmarking harness to know whether the chosen format actually met their size/quality bar. None of this is conceptually difficult, but all of it is friction that falls disproportionately on

practitioners who are domain experts in the model they are deploying rather than experts in the quantization tooling itself.

QuantForge collapses this friction into a single application. It provides one interface for two quantization ecosystems, parses model architecture directly from file headers so users see parameter count, layer count, and hidden dimension before committing to a format, and — critically — pairs every quantization run with a benchmark or quality-metrics panel so the tradeoff being made is visible rather than assumed.

02 Background and Related Work

2.1 Post-Training Quantization for Language Models

Post-training quantization (PTQ) reduces the numerical precision of a trained model's weights without retraining, in contrast to quantization-aware training which bakes precision reduction into the training loop itself [2]. PTQ is the dominant approach for open-weight LLM deployment because it requires no access to the original training pipeline or data — only the released weights.

Naive PTQ (round-to-nearest, RTN) simply rounds each weight to the nearest representable value at the target bit width. RTN is fast and requires no calibration data, but degrades disproportionately at 4-bit and below because it treats every weight as equally important. Two calibration-aware refinements dominate current practice:

GPTQ performs layer-wise reconstruction using an approximate second-order (Hessian) error correction, updating remaining weights in a layer to compensate for the rounding error already introduced in previously quantized weights [3]. This produces measurably lower perplexity increases than RTN at the same bit width, at the cost of requiring a calibration pass over representative data.

AWQ (Activation-aware Weight Quantization) instead identifies the small subset of weight channels that activations depend on most strongly and preferentially preserves their precision through per-channel scaling, leaving the bulk of the weight matrix at low precision [4]. AWQ typically achieves better quality retention than GPTQ at equivalent bit width for a comparable calibration cost, which is the basis for QuantForge defaulting its MLX pipeline to AWQ.

2.2 Two Incompatible Ecosystems

GGUF (the file format used by `llama.cpp`) and MLX's SafeTensors-based checkpoint format emerged from different projects with different design priorities, and neither reads the other natively. GGUF's `K_M`-suffixed quantization types (e.g., `Q4_K_M`) use per-block scale factors with mixed bit-widths across a tensor's blocks, tuned empirically over `llama.cpp`'s development history for a good size/quality tradeoff without calibration data [5]. MLX's quantization path, built for Apple's array framework, applies uniform per-group bit-width reduction (commonly 4- or 8-bit, at a configurable group size) and is the format `mlx-lm` expects for GPU-accelerated inference on Apple Silicon [6].

A practitioner deploying to `llama.cpp`-based runtimes (llama.cpp itself, Ollama, LM Studio's GGUF backend) needs GGUF; a practitioner deploying to native Apple Silicon inference via MLX needs the MLX format. Many practitioners need both — a GGUF build for portability and an MLX build for the fastest possible on-device throughput. Historically this has meant maintaining two separate toolchains, each with its own installation requirements, invocation syntax, and quality-verification process. QuantForge is, to the authors' knowledge, the first application to present both pipelines behind one consistent interface.

2.3 The Bandwidth-Bound Nature of LLM Inference

Autoregressive LLM decoding generates one token at a time, and each step must read the model's entire weight set from memory. On modern hardware, the arithmetic required per token is small relative to the memory bandwidth required to stream the weights, which means decode-time throughput is bandwidth-bound, not compute-bound, for the great majority of consumer and prosumer hardware [7]. This is precisely why quantization — which reduces the number of bytes that must be streamed per token — produces near-linear speedups rather than merely saving disk space: a 4-bit model does not just occupy a quarter of the memory of its 16-bit source, it can generate tokens several times faster on the same hardware, because there are proportionally fewer bytes to move per step.

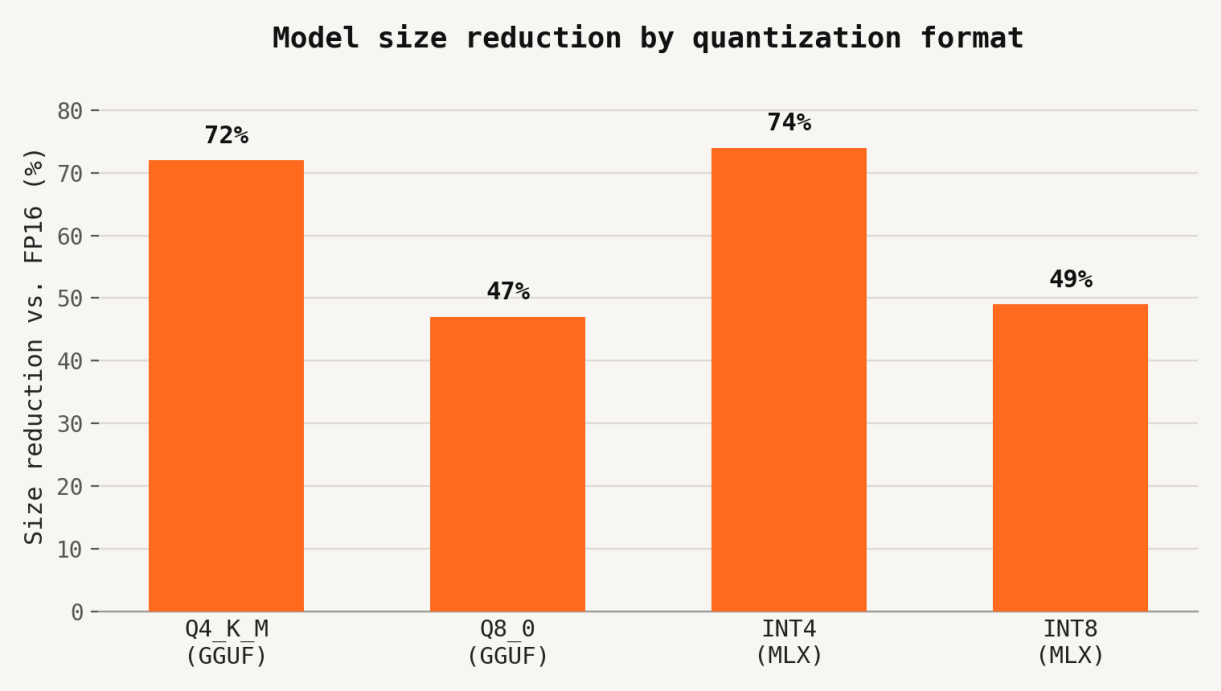

Figure 1. Size reduction achieved by each QuantForge quantization format, relative to FP16, derived from published bits-per-weight ratios for GGUF and MLX.

This relationship is the empirical basis for QuantForge's benchmark panel: tokens/sec, memory footprint, and file size are reported together because they move together, and a practitioner evaluating a quantization format needs to see that correlation, not just the size delta.

03 Architecture

3.1 Dual-Pipeline Design

QuantForge's `ModelViewModel` orchestrates two independent backend services against a shared model library: `QuantizationService`, which shells out to a located `llama-quantize` binary for GGUF targets, and `MLXQuantizationService`, which invokes `python3 -m mlx_lm.convert` as a subprocess for SafeTensors and PyTorch targets. Both services stream subprocess output

back to the UI as structured progress events rather than raw text, so the app can show a live log while also updating a determinate or indeterminate progress indicator.

Binary and interpreter discovery is deliberately defensive: QuantForge searches Homebrew, MacPorts, and standard system paths for `llama-quantize`, and searches Homebrew, pyenv, miniforge, miniconda, and mambaforge paths for a Python 3 interpreter with `mlx-lm` installed. When discovery fails, the app surfaces a banner with the exact remediation command (e.g., `brew install llama.cpp`, `pip install mlx-lm`) rather than a bare error, and offers a manual "Locate Binary…" picker as a fallback — treating missing native dependencies as an expected, recoverable state rather than an exception.

3.2 Metadata-First Model Inspection

Before a user commits to a quantization format, QuantForge's `MetadataReader` parses the model's architecture directly from its file headers: GGUF's binary key-value header for `.gguf` files, and `config.json` / `model.safetensors.index.json` for SafeTensors and PyTorch directories. Parameter count, layer count, hidden dimension, and vocabulary size are extracted without loading the weight tensors themselves, which keeps inspection fast even for multi-gigabyte checkpoints. This gives the user the context needed to make an informed format choice — a 1B-parameter model and a 70B-parameter model warrant different bit-width defaults — before any quantization work begins.

3.3 Format Selection Model

The GGUF pipeline exposes `Q4_K_M` (the general-purpose default, ~4.5 bits/weight, roughly 0.28× the FP16 size) and `Q8_0` (near-lossless, ~8.5 bits/weight, roughly 0.53× the FP16 size), reflecting `llama.cpp`'s own recommended defaults [5]. The MLX pipeline exposes three independent axes — method (`GPTQ` / `AWQ`), bit width (`INT4` / `INT8`), and group size (`G32` / `G64` / `G128`) — plus a calibration corpus selector (`None`, a bundled general-purpose corpus, or user-supplied custom text). Each axis is modeled as its own Swift enum (`MLXQuantMethod`, `MLXBitsPerWeight`, `MLXGroupSize`) so the view layer can render every valid combination without special-casing, and so the estimated-outcome calculations described in §3.4 stay in sync with whatever the user selects.

3.4 Benchmarking and Quality Estimation

On completion of a GGUF quantization run, QuantForge's `BenchmarkResult` reports tokens/sec, peak memory, perplexity, and file size for both the original and quantized model side by side, with

derived size-reduction, speedup, memory-reduction, and perplexity-increase percentages computed directly from the measured values. On completion of an MLX run, `MLXQualityMetrics` reports the actual output size and reduction percentage alongside an estimated speedup and estimated perplexity delta — the latter derived from published bit-width and calibration-method benchmarks (a baseline per-bit-width perplexity increase, discounted roughly 30% under GPTQ calibration or roughly 40% under AWQ calibration) rather than a fresh perplexity evaluation, since computing perplexity on a held-out set is materially more expensive than the quantization step itself. QuantForge labels these figures `calibrated` or `RTN` so the distinction between a measured result and a literature-derived estimate is never ambiguous to the user.

3.5 Universal SwiftUI Delivery

QuantForge is built as a single SwiftUI codebase targeting iOS/iPadOS 16 and macOS 13, using `NavigationSplitView` to adapt between a compact iPad layout and a full macOS multi-column window. Model import, metadata inspection, and library browsing work identically on both platforms; the quantization backends themselves require macOS, since both `llama-quantize` and `mlx-lm` are native/Python binaries with no iOS-compatible equivalent. This mirrors the platform-conditional pattern used elsewhere in the authors' Apple Silicon ML tooling, where training- and compute-heavy operations are gated to macOS while inspection and review surfaces remain universal.

04 Economic and Workflow Analysis

4.1 The Cost of Format Ambiguity

Choosing the wrong quantization format is not merely a quality-of-life issue; it has direct, measurable cost. A model quantized more aggressively than necessary produces a visibly worse user experience for the eventual application (elevated perplexity, more frequent generation artifacts); a model quantized more conservatively than necessary wastes memory and disk that could otherwise support a larger context window or a second loaded model. Because both errors are only discoverable after quantization and evaluation, and because a naive terminal-only workflow provides no side-by-side comparison, practitioners without a benchmarking habit tend to anchor on whatever default format they saw in a tutorial rather than the format appropriate to their deployment target. QuantForge's benchmark and quality panels convert this from a one-shot,

invisible decision into an inspectable one: every quantization run produces the comparison data needed to decide whether to try a different format, at no additional cost beyond the run that already had to happen.

4.2 Time-to-Deployed-Model

A practitioner assembling a GGUF and an MLX build of the same model by hand needs, at minimum, two correctly-installed toolchains, two correctly-recalled command syntaxes, and a self-written benchmarking script to compare results — realistically 30–60 minutes of setup and scripting for a first-time workflow, repeated per model family as tooling and flags drift. QuantForge reduces this to two menu-driven runs against a pre-installed dependency set, with benchmarking built in rather than bespoke. For a practitioner iterating across several candidate formats for a single deployment — a common pattern when the "right" bit width is not known in advance — this compounds: the marginal cost of trying a second or third format drops from another round of manual scripting to a dropdown change and a re-run.

4.3 Local-First Economics

Quantization is a one-time, CPU/GPU-bound batch operation, not a repeated inference workload, so its cloud-cost profile differs from the inference-serving case: the relevant comparison is not per-call API pricing but the cost of a cloud GPU instance rented specifically to run a conversion script, plus the upload/download time for multi-gigabyte weight files. Running the identical operation on Apple Silicon hardware the practitioner already owns eliminates both the instance rental and the network transfer, at the marginal electricity cost of a compute-bound task lasting minutes, consistent with the on-device economics argument developed for training workloads in the authors' companion analysis of local ML tooling.

05 Benefits and Impact

5.1 Lowering the Floor for Local LLM Deployment

Running a quantized open-weight model locally — via `llama.cpp`, Ollama, LM Studio, or an MLX-based application — no longer requires the practitioner to also become proficient in producing the quantized weights themselves. QuantForge separates "I know what model I want to run locally"

from “I know how to operate `llama-quantize` and `mlx_lm.convert` from a terminal,” which is a meaningfully lower bar and opens local deployment to a broader population of hobbyists, researchers, and engineers outside the ML infrastructure specialty.

5.2 Making the Quality Tradeoff Legible

Because every quantization run in QuantForge produces a size/speed/quality readout rather than just a file, the application turns an implicit tradeoff into an explicit, comparable one. A user who quantizes the same model at `Q4_K_M` and `Q8_0` in succession can see the actual perplexity and size difference between them, rather than relying on generic guidance (“Q4 is usually fine”) that may or may not hold for their specific model and use case.

5.3 Bridging Two Deployment Targets from One Workflow

Because QuantForge supports both GGUF and MLX from the same library and the same imported model, a practitioner who wants a portable GGUF build for distribution and a native MLX build for their own Apple Silicon inference no longer needs to context-switch between two toolchains and two mental models to produce both. This is particularly relevant for the growing set of applications — including the authors’ own on-device inference tooling — that support multiple backend formats and benefit from having both available without a second, separate conversion effort.

5.4 Reproducibility and Dependency Transparency

By surfacing exactly which binary and which Python environment it is invoking, and by failing with actionable remediation rather than a silent or cryptic error, QuantForge makes the quantization environment legible rather than opaque. This matters for reproducibility: a benchmark result is only meaningful if the practitioner can state with confidence which backend version and configuration produced it, and QuantForge’s explicit dependency-discovery banner is the mechanism that makes that traceable rather than assumed.

06 Limitations and Future Work

Estimated versus measured MLX quality metrics. The MLX pipeline’s perplexity-delta figures are derived from published per-bit-width, per-calibration-method literature benchmarks rather than a

fresh evaluation against the user's specific model and data, because a full perplexity pass is computationally expensive relative to the quantization step itself. A future version could offer an optional, explicitly opt-in perplexity evaluation pass for users who need a measured rather than estimated figure.

macOS-only quantization backends. Because `llama-quantize` and `mlx-lm` are native/Python tools without iOS equivalents, the iPadOS build of QuantForge is limited to model import, metadata inspection, and library browsing. Users who want to quantize on iPad must still perform that step on a Mac, though the resulting library and benchmarks sync identically across both platforms.

Fixed format catalog. The current GGUF pipeline exposes two curated formats (`Q4_K_M`, `Q8_0`) rather than `llama.cpp`'s full quantization type catalog, and the MLX pipeline exposes `GPTQ` / `AWQ` with a fixed set of bit-widths and group sizes. This is a deliberate simplicity/coverage tradeoff for the initial release; expanding the exposed format matrix for advanced users is a natural extension.

Single-machine workflow. QuantForge currently operates against models and backends local to the machine it runs on. Offloading quantization of very large models to a more capable remote machine — a pattern already used elsewhere in the authors' Apple Silicon tooling for heavy training workloads — is a plausible extension for models too large to comfortably quantize on the local device.

07 Conclusion

QuantForge demonstrates that LLM quantization — a task that has remained command-line-only even as the surrounding ecosystem of local inference applications has become approachable and graphical — can be delivered as a native, universal Apple application without sacrificing access to the two backends (`llama.cpp` and MLX) that matter most for on-device deployment. By pairing metadata-driven model inspection with mandatory before/after benchmarking, QuantForge converts quantization from a one-shot, faith-based terminal command into an inspectable, comparable decision. As the population of practitioners running open-weight models locally continues to grow faster than the population fluent in quantization tooling, applications that close this specific gap — turning an infrastructure skill into a guided workflow — are positioned to matter well beyond quantization itself.

References

[1] Kim, S., et al., “SqueezeLLM: Dense-and-Sparse Quantization,” arXiv:2306.07629, 2023.

[2] Gholami, A., et al., “A Survey of Quantization Methods for Efficient Neural Network Inference,” arXiv:2103.13630, 2021.

[3] Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D., “GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers,” arXiv:2210.17323, 2022.

[4] Lin, J., et al., “AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration,” arXiv:2306.00978, 2023.

[5] Gerganov, G., et al., “llama.cpp: LLM inference in C/C++,” GitHub, 2023–2026. https://github.com/ggml-org/llama.cpp

[6] Apple Inc., “MLX: An array framework for Apple Silicon,” GitHub, 2023–2026. https://github.com/ml-explore/mlx

[7] Kim, S., et al., “Full Stack Optimization of Transformer Inference: a Survey,” arXiv:2302.14017, 2023.