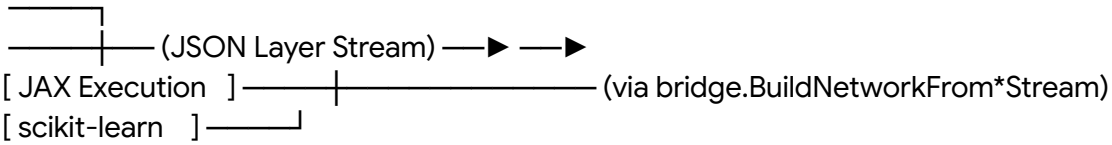
Comprehensive Analysis of Multi-Engine Weight Absorption and Edge-Native Retraining within Loom's Planetbridging Framework

The landscape of edge artificial intelligence is undergoing a significant transition from cloud-dependent inference pipelines to fully sovereign, local execution.[1] Historically, deploying neural networks to resource-constrained devices has been a highly fragmented process.[3] Each major AI runtime operates as an isolated ecosystem, characterized by its own closed file formats, custom operator dialects, and hardware-specific math libraries.[3] This fragmentation often forces developers to perform lossy model conversions, which introduce implementation bugs, high maintenance overhead, and cumulative numerical drift.[4]

Loom's Planetbridging framework addresses this challenge by providing a universal runtime bridging layer.[3] Rather than compiling static, intermediate file-based representations, Loom uses an in-memory virtualization pipeline designed to absorb live model parameters directly from disparate host frameworks.[3] This report evaluates the architectural viability of Loom's multi-engine weight absorption, analyzes its native edge-training capabilities, and compares its design to other contemporary edge AI engines.

The Architectural Viability of Multi-Engine Weight Absorption

The core technical challenge in edge deployment is the structural consolidation of model weights developed across heterogeneous training environments.[3] Traditional compilation frameworks treat weight migration as an offline, ahead-of-time translation process, which is highly sensitive to semantic gaps between the source engine's abstract syntax tree and the target engine's operator set.[4] Loom's Planetbridging addresses this by introducing a live, in-memory streaming protocol that can dynamically ingest active weights from up to four distinct source engines—PyTorch, TensorFlow, JAX, and scikit-learn—and consolidate them into a single, cohesive runtime entity.[1]



The ingestion process begins inside the native python execution environment where the source model is running.[3] Instead of saving the network to disk, a lightweight python helper library captures the active, in-memory weight matrices.[3] It serializes these parameters and the network topology into a JSON layer stream, which is broadcast to a local network interface via POST /api/v1/loom/stream/*.[3] On the receiving end, Loom's native Go-based comparison host catches the stream and uses localized builder methods, such as bridge.BuildNetworkFrom*Stream, to reconstruct the network architecture and compile it into a unified, binary .entity checkpoint.[1]
This streaming approach is validated across twelve standard bedrock layers [3]:

Bedrock Layer ID	Source Engine Coverage	Loom Stream Status	Loom Compare Status	Empirical Tolerance Threshold
Dense	PyTorch, TensorFlow, JAX, scikit-learn [3]	PASS [3]	43/48 PASS [3]	Single-precision fp32 ($<$) [3]
CNN1 / CNN2 / CNN3	PyTorch, TensorFlow, JAX [3]	PASS [3]	12/12 PASS [3]	Single-precision fp32 ($<$) [3]
Multi-Head Attention (MHA)	PyTorch, TensorFlow, JAX [3]	PASS [3]	12/12 PASS [3]	Single-precision fp32 ($<$) [3]
LSTM	PyTorch, TensorFlow, JAX [3]	PASS [3]	12/12 PASS [3]	Single-precision fp32 ($<$) [3]
Simple RNN	PyTorch, TensorFlow, JAX [3]	PASS [3]	12/12 PASS [3]	Single-precision fp32 ($<$) [3]
LayerNorm	PyTorch, TensorFlow, JAX [3]	PASS [3]	24/24 PASS [3]	Single-precision fp32 ($<$) [3]

Embedding	PyTorch, TensorFlow, JAX [3]	PASS [3]	24/24 EXACT [3]	Bit-exact bitwise parity (0.0) [3]
RMSNorm	PyTorch, TensorFlow, JAX [3]	PASS [3]	24/24 PASS [3]	Single-precision fp32 ($<$) [3]
SwiGLU	PyTorch, TensorFlow, JAX [3]	PASS [3]	24/24 PASS [3]	Single-precision fp32 ($<$) [3]
Residual	PyTorch, TensorFlow, JAX [3]	PASS [3]	24/24 PASS [3]	Single-precision fp32 ($<$) [3]
Mixer v1	PyTorch, TensorFlow, JAX [3]	PASS [3]	3/3 PASS [3]	Single-precision fp32 ($<$) [3]
Mixer v2	PyTorch, TensorFlow, JAX [3]	PASS [3]	3/3 POC PASS [3]	Floating-point limit ($\sim 5 \times$) [3]

The physical pipeline of streaming a live model and comparing its output to the original host engine is illustrated by running 01_hello_stream.py on a dense MLP model (mlp_32_64_32_16_8_relu with 4 layers).[3] This script executes a forward pass on PyTorch, streams the live weights to the comparison host, and saves the binary .entity file directly to the disk.[3] The outputs generated by PyTorch and Loom's runtime are then validated against a shared test vector, matching within standard single-precision tolerances.[3] The same verification pipeline successfully validates multi-head attention layers, such as mha_8_2_4, confirming that multi-engine parameter absorption is mathematically valid and structurally sound.[3]

Active Edge Training and the Deterministic Neural Virtual Machine

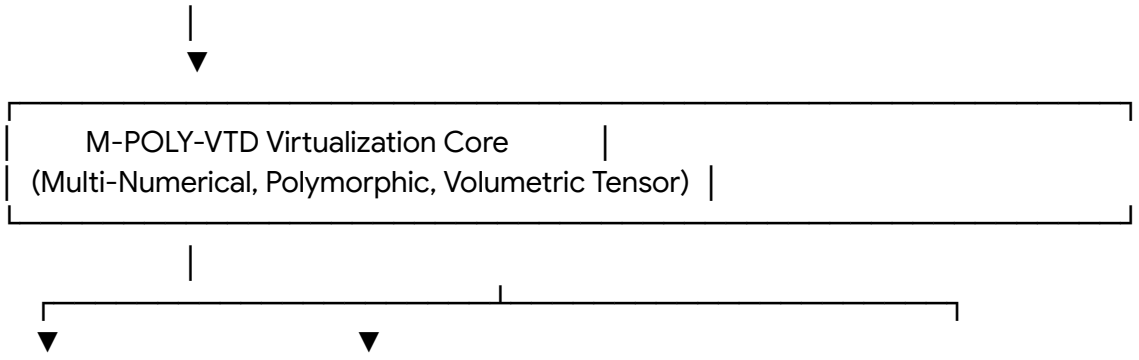
Consolidating multi-engine weights into a single execution target is only the initial phase; the subsequent architectural requirement is enabling active training within that same runtime.[1]

Loom is designed as a pure-Go Deterministic Neural Virtual Machine (DNVM).[1] This architecture decouples execution from system-level platform variations, allowing backpropagation and weight optimization to occur natively on the target device without frozen dependencies.[1]

The Prerequisite of Bit-Exact Cross-Platform Determinism

In standard deep learning runtimes, on-device training is highly susceptible to mathematical divergence.[4] Floating-point operations executed on disparate physical architectures (such as ARM processors versus NVIDIA CUDA GPUs) typically yield minor numerical variations due to compiler optimizations, platform-specific math libraries, and hardware-specific rounding behaviors.[5] During inference, these micro-divergences are generally negligible; however, during backpropagation, cumulative mathematical drift across multiple training iterations can cause the gradient calculations to diverge, ultimately degrading the model.[5]

The DNVM architecture addresses this by enforcing bit-exact cross-platform determinism.[1] By utilizing a pure-Go execution core with explicit, deterministic floating-point operations, Loom guarantees a Mean Absolute Error (MAE) of less than 10^{-8} across WebAssembly, Android, iOS, Windows, macOS, and Linux.[1] This numerical stability ensures that any gradient updates calculated during edge training remain mathematically consistent regardless of the underlying hardware.[1]



The M-POLY-VTD Engine and Multi-Numerical Training

Loom's execution environment treats neural networks as 3D spatial grids where individual cells support dynamic precision adjustments, rather than as sequential layers.[7] This is powered by its M-POLY-VTD (Multi-Numerical Architecture, Polymorphic Layer-Morphing, Volumetric Tensor Dispatch) virtualization core.[7]

The DNVM includes a native data-type system supporting 21 distinct numerical types, ranging from double-precision $fp64$ down to 1-bit binary representations.[1] During on-device training, the engine can dynamically morph the precision of individual layers to optimize for local hardware features—such as hardware-level ternary operations—without altering the global model structure.[1] This allows the training pass to run directly within the virtual machine, avoiding memory bandwidth bottlenecks on edge devices.[1]

Backpropagation and Localized Target Propagation

To facilitate training on edge devices, the DNVM provides a dual-path optimization architecture [1]:

1. **Direct Backpropagation:** Loom implements standard gradient-descent optimization passes directly in pure Go, allowing full parameter updates across all standard layers.[1]
2. **Neural Target Propagation (Tweening):** As an alternative to traditional backpropagation, Loom's core engine includes a localized target propagation system implemented natively via tween.go.[6] Tweening replaces global gradient chains with localized, gap-based Hebbian learning that updates layer parameters based on local activation targets rather than global error distribution.[6] This reduces the peak memory footprint during training because the device does not need to store the massive global activation graphs required by traditional backpropagation.

Technical Verification and the Welvet Ladder

The operational integrity of this weight ingestion and local retraining pipeline is validated using a multi-phase testing framework.[3] Inside the planetbridging directory, validation scripts run continuous comparisons between the source host and Loom's environment.[3]

The Three-Way Validation Pipeline

The primary testing tool is the Welvet Ladder (implemented in 05_welvet_ladder.py), which conducts a three-way output validation [3]:

$$\text{Native PyTorch Output} \quad \longleftrightarrow \quad \text{Loom-Stream Output} \quad \longleftrightarrow \quad \text{Welvet Reloaded Enti}$$

This ladder validates the model's performance from its native execution, through the live in-memory stream, and after being reloaded from the binary .entity file.[6] For standard structures like convolutional layers (CNN1), multi-head attention (MHA), and layer normalization, the output remains stable throughout the entire pipeline.[3]
However, the three-way validation identifies specific edge cases where the reloading pipeline can encounter challenges.[3] Standard layers such as LSTM, Simple RNN, and deep mixer networks are noted as having C Application Binary Interface (C ABI) gaps.[3] These gaps can introduce minor parsing discrepancies during the raw reload phase, so the testing harness is

designed to bypass these layers during the Welvet reload check while still validating them during the live streaming phase.[3]

Comprehensive Framework Comparison

Evaluating Loom's Planetbridging requires comparing its architectural design with other edge inference and training runtimes.[4] The primary modern alternatives are Microsoft's ONNX Runtime (ORT) On-Device Training [8], Google's LiteRT (formerly TensorFlow Lite) [9], Meta's ExecuTorch [2], and Apple's Core ML.[10]
The table below provides a detailed comparison of these runtimes:

Architectural Dimension	Loom (with Planetbridging)	ONNX Runtime (ORT)	LiteRT (TensorFlow Lite)	ExecuTorch	Apple Core ML
Ingestion Paradigm	**Dynamic Stream:** Extracts weights from active python execution memory.[3]	**Static Compile:** Generates intermediate exchange graphs ahead-of-time.[4]	**Static Compile:** Converts source models into flatbuffer representations.[9]	**Static Compile:** Graph-level compilation into a unified executable.[2]	**Static Compile:** Converts model structures using custom tools.[10]
Multi-Engine Support	**Unified Hub:** Ingests PyTorch, TF, JAX, and scikit-learn.[3]	**Broad Compatibility:** Parses standard ONNX files exported from major libraries.[4]	**Conversion-Locked:** Optimized for TF; requires conversion tools for other frameworks.[9]	**Ecosystem-Locked:** Limited to PyTorch export paths.[2]	**Conversion-Locked:** Restricted to models translated via coremltools.[10]
In-Memory Absorption	**Supported:** Real-time JSON stream compiled into binary .entity files.[3]	**Unsupported:** Relies on offline export configurations.[4]	**Unsupported:** Requires pre-compiled flat files.[9]	**Unsupported:** Bound to compiled static .pte model graphs.[2]	**Unsupported:** Bound to compiled directory structures.[1][3]

Edge Training Approach	**Unified Engine:** Native backpropagation and target propagation in-engine.[1]	**Split Phase:** Generates training artifacts offline before edge execution.[8]	**Signature-Based:** Exposes trainable weights via explicit execution signatures.[1][2]	**Orchestrated:** Integrated with NVIDIA FLARE for federated deployments.[17]	**Layer-Locked:** Restricts training to layers flagged as updatable.[18]
Cross-Platform Parity	**Bit-Exact:** $MAE <$ across CPU, GPU, x86, ARM, and WebAssembly.[1]	**Variable:** Dependent on system-level libraries and hardware backends.[5]	**Variable:** Subject to platform-specific floating-point variations.[5]	**Variable:** Subject to hardware delegate execution pathways.[4]	**Hardware-Locked:** Optimized for Apple Silicon; differs from other architectures.[5]
Base Engine Footprint	**Ultra-Lightweight:** Single compiled binary of approximately $10\ \text{MB}$.[1]	**Heavyweight:** Typically $50–100\ \text{MB}$ engine footprint.[5]	**Moderate:** Embedded runtime is compact, but full setups exceed $500\ \text{MB}$.[5]	**Modular:** Base runtime footprint of approximately $50\ \text{KB}$.[2]	**System-Integrated:** Native operating system library with zero direct app footprint.[18]
Vendor Dependency	**Zero Lock-In:** Pure Go runtime with zero CGO or cloud dependencies.[1]	**Low-to-Medium:** Multi-platform support, but requires complex build pipelines.[5]	**Moderate:** Heavily integrated with Google's mobile and web platforms.[9]	**Ecosystem-Locked:** Bound to PyTorch compilation and runtime tools.[2]	**Platform-Locked:** Restricted to Apple hardware environments.[13]

Architectural Synthesis and the Future of Edge AI

Comparing these design approaches highlights the trade-offs between compile-time optimization and runtime virtualization in edge computing.

The Trade-Offs of Native Graph Compilation

Runtimes like ExecuTorch and Apple Core ML achieve high performance on specialized hardware by using ahead-of-time (AOT) graph-level partitioning and compiling the network directly into hardware-specific binary instructions.[2] This optimization, however, creates platform lock-in.[4]

If a compiled model must run on an unsupported device or if a custom layer cannot be mapped to the target hardware's operator set, execution reverts to a slow CPU fallback or fails entirely.[4] Additionally, on-device training in these compiled frameworks is often constrained; Core ML, for example, requires specific layers to be designated as "updatable" before deployment, limiting the model's ability to adapt dynamically.[18]

The Virtualization Strategy

Loom's DNVM bypasses compile-time constraints by using a virtualization strategy.[1] By treating the virtual machine itself as the target runtime and building it in pure Go with zero system-level dependencies, Loom achieves cross-platform consistency across diverse hardware configurations.[1]

The Planetbridging framework (v0.5.0) demonstrates this flexibility by dynamically streaming active weights from multiple source engines directly into the DNVM, avoiding the need for static file conversion chains.[1] Combined with built-in backpropagation and memory-efficient target propagation, this design provides a viable path for training and optimizing models natively on edge devices.[1]

Future Development and Roadmap

The current release of Planetbridging (v0.5.0) marks the completion of the ingestion phase, enabling live parameter streaming from host platforms to Loom's deterministic runtime.[3] The project's development path is structured around several upcoming milestones:

- **Phase B (v1.0.0):** Establishing bidirectional bridging by implementing native exporters from Loom to ONNX, Safetensors, and GGUF, enabling models to flow back out to engines like llama.cpp and CoreML.[3]
- **Phase C (v1.x):** Introducing offline file-based ingestion directly into the comparison host, allowing direct loads of Safetensors, GGUF, and ONNX files without requiring an active Python environment.[3]
- **Phase D (v1.x to 2.0):** Adding support for more specialized layer types (such as transpose convolutions and custom softmax variants), expanding training capabilities, and optimizing edge execution.[3]

By addressing the challenges of cross-platform determinism and multi-engine weight consolidation, Loom's Planetbridging framework provides a practical solution for deploying,

running, and updating models locally at the edge.[1]

Works cited

1. Loom - Layered Omni-architecture Openfluke Machine - GitHub, accessed July 1, 2026, https://github.com/openfluke/loom
2. ExecuTorch - On-Device AI Inference Powered by PyTorch, accessed July 1, 2026, https://executorch.ai/
3. README.md
4. ExecuTorch -- A Unified PyTorch Solution to Run AI Models On-Device - arXiv, accessed July 1, 2026, https://arxiv.org/pdf/2605.08195
5. LOOM: The Universal AI Runtime That Works Everywhere (And Why That Matters) - Medium, accessed July 1, 2026, https://medium.com/@planetbridging/loom-the-universal-ai-runtime-that-works-everywhere-and-why-that-matters-54de5e7ec182
6. loom/README.md at main · openfluke/loom - GitHub, accessed July 1, 2026, https://github.com/openfluke/loom/blob/main/README.md
7. M-POLY-VTD: Architecture Overview — Loom Docs - OpenFluke, accessed July 1, 2026, https://openfluke.com/docs/overview
8. On-Device Training | onnxruntime, accessed July 1, 2026, https://onnxruntime.ai/docs/get-started/training-on-device.html
9. LiteRT: High-Performance On-Device Machine Learning Framework | Google AI Edge, accessed July 1, 2026, https://developers.google.com/edge/litert
10. Updatable Models — Guide to Core ML Tools - Apple, accessed July 1, 2026, https://apple.github.io/coremltools/docs-guides/source/updatable-model-examples.html
11. Training - ONNX Runtime, accessed July 1, 2026, https://onnxruntime.ai/training
12. Training AI Models Directly on Android: A Practical Guide Using LiteRT + TensorFlow Signatures | by Lubna Mariyam | Medium, accessed July 1, 2026, https://medium.com/@lubnamariyam62/training-ai-models-directly-on-android-a-practical-guide-using-litert-tensorflow-signatures-d959c66d202e
13. Personalizing a Model with On-Device Updates | Apple Developer Documentation, accessed July 1, 2026, https://developer.apple.com/documentation/coreml/personalizing-a-model-with-on-device-updates
14. ExecuTorch Concepts - PyTorch documentation, accessed July 1, 2026, https://docs.pytorch.org/executorch/0.4/concepts.html
15. Overview - Python API documentation - ONNX Runtime, accessed July 1, 2026, https://onnxruntime.ai/docs/api/python/on_device_training/overview.html
16. On-device training in TensorFlow Lite, accessed July 1, 2026, https://blog.tensorflow.org/2021/11/on-device-training-in-tensorflow-lite.html
17. Effortless Federated Learning on Mobile with NVIDIA FLARE and Meta ExecuTorch, accessed July 1, 2026, https://developer.nvidia.com/blog/effortless-federated-learning-on-mobile-with-nvidia-flare-and-meta-executorch/

18. What's new in Core ML 3 - Fritz ai, accessed July 1, 2026, https://fritz.ai/whats-new-in-core-ml/
19. On-Device Training: Efficient training on the edge with ONNX Runtime | Microsoft Open Source Blog, accessed July 1, 2026, https://opensource.microsoft.com/blog/2023/05/31/on-device-training-efficient-training-on-the-edge-with-onnx-runtime/