Blueprint: Delta-Driven Dual-Stream LLM (8GB Variant)

Author: Frustrated AI Developer, 2024

# Executive Summary

We are refocusing the continual-learning architecture on budget hardware: a single RTX 3070 Ti (8 GB VRAM). The goal remains to explore delta-driven, online learning, but every component now assumes tight memory and throughput limits.

# Motivation and Constraints

* We want architectural insights, not state-of-the-art benchmark runs.
* Hardware budget is capped at a single consumer GPU with 8 GB VRAM.
* Setup must run comfortably inside VS Code terminals on Windows or Linux desktops.
* Inference and training loops must stay lightweight, modular, and debuggable.

# Strategic Objectives

* Produce a reference implementation that streams data and learns from prediction deltas without exhausting 8 GB VRAM.
* Keep the code modular so backbone swaps (TinyLlama, Phi-3 Mini, etc.) are trivial.
* Emphasize observability and reproducible experiments from inside VS Code.
* Document every step to invite collaboration from engineers with limited hardware.

# Hardware-Aware Design Choices

* Backbone: TinyLlama/TinyLlama-1.1B-Chat-v1.0 in 4-bit (bnb) or microsoft/Phi-3-mini-4k-instruct in 4-bit; both stay under roughly 6.5 GB during inference.
* Adapters: LoRA rank 4-8 targeting attention and MLP blocks only; freeze everything else.
* Batching: micro-batch of 1 sequence, gradient accumulation to reach effective context sizes.
* Context window capped at 1024-2048 tokens to avoid KV cache spikes on 8 GB cards.
* Checkpointing and CPU offload ready toggles if we explore heavier backbones.

# Architecture Overview (Core Principles)

We keep the delta-learning recipe: predicted next frame versus delayed reality, two-tick buffer, EMA teacher, and uncertainty gating. The difference is careful budgeting of compute and memory.

* Streaming backbone with smaller hidden size but persistent state.
* Forward predictor head remains lightweight; optionally share weights with LM head to save parameters.
* Two-tick buffer stores logits and minimal metadata only; avoid retaining full activations.
* EMA teacher mirrors the predictor head (not the entire trunk) to keep costs low.
* Uncertainty gate uses entropy only, no expensive ensembles.

# Implementation Adjustments

* Quantization defaults to 4-bit NF4; fallback to 8-bit if local kernels misbehave.
* Tokenizer loads once and is reused across training and chat to minimize host RAM.
* Training loop detaches tensors aggressively to prevent graph buildup.
* Replay buffer stores tensors on CPU to free VRAM.
* Logging is concise by default but DEBUG mode reveals full details for diagnosis.

# Resource Requirements

* GPU: RTX 3070 Ti 8 GB (desktop or laptop).
* CPU: 6 or more cores recommended for tokenization and JSONL streaming.
* RAM: 16 GB recommended, 32 GB comfortable.
* Storage: less than 10 GB for models, caches, and logs.

# Workflow (VS Code Focus)

* Use the VS Code integrated terminal (Ctrl+`).
* Provide .env.example with Hugging Face token placeholder; source it in VS Code terminal.
* Offer .vscode/launch.json snippets for chat and training scripts.
* Pin dependencies in requirements.txt for repeatable installs.

# Phase Roadmap (8 GB Edition)

## Phase A: Bring-up (2-3 days)

* Download TinyLlama 1.1B in 4-bit; verify load and chat inference.
* Run streaming trainer on a toy JSONL to validate buffer and gate on GPU.
* Record VRAM usage via torch.cuda.memory\_allocated() logs.

## Phase B: Delta Learning Enhancements (1 week)

* Add residual and consistency losses with careful detaches to avoid large graphs.
* Introduce prioritized replay stored on CPU, transferring only mini-batches to GPU.
* Benchmark throughput versus VRAM; adjust sequence length to keep peaks under 7.5 GB.

## Phase C: Tooling and Evaluation (1 week)

* Integrate simple dashboards (TensorBoard or VS Code notebooks).
* Add unit and integration tests runnable under pytest inside the VS Code terminal.
* Prepare optional CPU-only fallback for collaborators without GPU.

# Success Metrics

* Residual cross-entropy drops faster than vanilla LoRA finetune under identical token budgets.
* Max VRAM usage stays under 7.5 GB during both training and chat.
* Response latency stays below 2 seconds for 128-token generations.
* Code remains readable and runnable inside VS Code with minimal setup.

# Risks and Mitigations

* Bitsandbytes instability on local drivers -> provide CPU fallback guidance and CUDA toolkit notes.
* VRAM spikes due to long sequences -> add dynamic truncation and VRAM logging guardrails.
* Architecture drift -> schedule audits and keep doc/code changes synchronized.

This lean blueprint keeps the architectural ambitions intact while respecting the realities of an 8 GB GPU. We sacrifice raw scale, not curiosity.