LYRN: A Technical Framework for Symbolic, Memory-Based Cognition in Local I

Abstract

This document expands on the core architecture of LYRN, the Living Yield Relational Network, by detailing its runtime behavior, execution model, and deployment design. Where the primary whitepaper introduces LYRN's cognitive framework and symbolic memory structure, this technical guide explains how the system runs in practice. It outlines how LYRN loads structured memory snapshots directly into RAM, applies field-level deltas without prompt repetition, and operates continuously through a direct-inference architecture that bypasses servers and middleware.

Key components such as the heartbeat cycle, relational memory tables, and KV cache reuse are described in depth, showing how they work together to produce long-term continuity and adaptive presence without increasing model size or token cost. The result is a lightweight, modular system that transforms local language models into persistent cognitive agents. This supplement is intended for engineers, implementers, and researchers seeking a practical blueprint for building symbolic, memory-based AI systems on consumer hardware.

1. Purpose of This Document

This document serves as a technical supplement to the primary LYRN whitepaper, expanding on the execution model, memory runtime behavior, and hardware performance considerations of the LYRN framework.

Where the original whitepaper focused on cognitive architecture, symbolic reasoning, and the philosophical foundations of LYRN, this guide explains how the system actually runs. It details what memory is loaded, when it is updated, and how the local LLM reuses cached context to simulate continuity and presence.

This runtime overview is intended for implementers, engineers, and researchers interested in:

Understanding how LYRN operates without traditional prompt injection

Building efficient local agents using symbolic memory

Designing modular snapshots and heartbeat cycles

Deploying LYRN on consumer hardware without cloud dependencies

The goal is to make LYRN understandable as a live system, not just a framework. By showing how memory flows, when cognition occurs, and where optimization happens, this guide bridges theory and practice.

2. Execution Architecture Overview

Overview of LYRN Components

LYRN functions as a modular runtime designed to manage memory, context, and conversational flow around a stateless language model. It is composed of discrete, tightly coordinated systems that handle memory injection, update logic, and behavior modulation through external control. Each component serves a specific role in maintaining long-term continuity and adaptive real-time reasoning, all without prompt reinjection.

The four primary components are:

Snapshot: A pre-tokenized, relationally structured memory map that defines static system context. It includes agent identity, instructions, user state, and prioritized memory tables.

Delta Layer: A lightweight update buffer containing field-level memory changes. Deltas are stored in individual files and tracked by a manifest. They overlay the snapshot to represent working memory.

Heartbeat Cycle: A reflective background loop that runs after each reasoning cycle. It processes the most recent interaction, evaluates updates, and generates deltas based on predefined relational rules.

Reasoning Cycle: The core interaction loop where the LLM receives user input, references memory, and generates output. It evaluates only new deltas and input, reusing the cached memory context to avoid reprocessing.

These four components operate in a continuous loop guided by the current system mode. Each layer will be described in detail in the following subsections.

Snapshot Memory

The snapshot is a relationally structured set of memory tables and fields that is injected at the start of every reasoning cycle. It includes components such as agent identity, user summaries, mode flags, and behavioral instructions. Because the snapshot is pre-tokenized and stored in the models KV cache, it can be reused efficiently without re-evaluation or token overhead.

Although the snapshot can be regenerated at any point, LYRN uses a hardware-aware strategy to time merges appropriately. In most cases, the size of the snapshot remains stable unless there are significant emotional shifts or the system gains new identity-level insights. This predictability allows the system to rebuild the snapshot without noticeable disruption. On slower hardware, LYRN can conceal the merge during TTS playback or insert a short message like give me a moment to think while preparing the update.

All snapshot merges occur after the reasoning cycle and before the heartbeat. This sequencing ensures state updates do not interfere with the user experience or model continuity.

At the foundation of the snapshot is the Relational Web Index, or RWI. This is a top-level memory table that defines the structure and traversal order of all included relational memory tables. It provides the LLM with a prioritized map of reasoning, beginning with critical context such as identity, user summaries, and active goals, then moving into secondary or mode-specific content.

A separate RWI Instructions table guides how the LLM processes each field listed in the index. These instructions specify how each field should be treated during processing, including whether it is referenced, ignored, accumulated, emotionally weighted, or considered eligible for delta updates. This design gives the model a structured, table-driven framework for reasoning across memory without relying on prompt repetition or token inference.

To support introspective processing, any memory table can optionally include a _hb_ column. This column holds field-specific instructions that apply only during the heartbeat cycle. Because most fields already have descriptive labels, the _hb_ layer introduces minimal token cost while enabling efficient post-turn memory updates.

This design enables both the reasoning and heartbeat cycles to operate from the same preloaded KV cache. As a result, LYRN maintains high memory efficiency and structural continuity across all turns, even on constrained hardware.

Delta Layer

The delta layer stores all temporary memory updates between snapshot merges. Each updated field is written to its own plain-text .txt file, using the field name as the filename. For example, trust_score.txt would contain only the current value for that field. If a field is updated multiple times during a session, each version is preserved. However, only the

most recent update is regarded as active.

LYRN does not delete or overwrite earlier deltas before a merge. Keeping previous versions ensures that the KV cache remains stable and prevents reprocessing of earlier content. The system injects only the latest active values based on a delta manifest, which tracks the current working set of updates.

The delta manifest is not seen by the model. It is maintained by the runtime and used to select the active deltas for both injection and merging. This keeps token overhead low while preserving clear field-level control over memory updates.

Only fields explicitly marked as delta-enabled in the snapshot may be updated in this way. This restriction protects structural and identity-critical memory. For special cases like insights, delta files are indexed, such as insight_139.txt, and accumulated rather than overwritten.

The models understanding of how to interpret deltas is based entirely on the RWI Instructions. These relational directives tell the LLM how each delta-enabled field behaves, how it should affect reasoning, and whether it should trigger summary updates or output filtering logic. This separation allows the system to maintain clean logic across both cycles without introducing redundant prompts or token inflation.

Heartbeat Cycle

The heartbeat cycle is triggered immediately after each reasoning cycle by injecting a single-line directive, such as mode: heartbeat. It operates silently in the background and does not produce a direct response to the user. Its primary function is to reflect on the most recent input and output, analyze conversational tone, update emotional or behavioral states, and prepare new field-level memory deltas.

These deltas are written to individual text files and registered in the delta manifest for use during the next reasoning cycle. The heartbeat uses both the RWI Instructions and any _hb_ fields to determine which memory fields should be processed, skipped, summarized, or reinforced. This enables efficient, field-specific updates without the need for additional prompts or memory injections.

Chat history is retained as a sequential stream of interactions, split into JSON-formatted logs that each hold up to 50 chat pairs. These episodic logs are never discarded unless manually reset. Each log includes a system-generated summary, a timestamp range, and metadata for each entry, allowing for granular lookup and episodic recall.

On higher-end systems, portions of these chat logs may be preloaded into the models KV cache to allow faster memory access and contextual blending during active sessions. On more limited hardware, only the latest entries and active deltas are retained in memory, while historical logs are stored externally and accessed on demand.

Reasoning Cycle

The reasoning cycle is the core interaction point between LYRN and the user. It is usually triggered by user input, but it can also be initiated internally when a follow-up is appropriate. For example, the system may respond again after a heartbeat reflection or when conversational pacing suggests a natural continuation. This flexibility allows LYRN to maintain more natural, human-like timing.

When input arrives, it is immediately recorded as a delta and labeled according to its source, such as user, heartbeat, or a scheduled system trigger. This delta is appended to the current set of active deltas and treated like any other memory update. The system then injects the pre-tokenized snapshot along with all active deltas, including the new input. Only these new delta values are evaluated by the model. All other memory content remains in the KV cache and is reused without reprocessing.

The model references field-level instructions defined in the relational memory structure to determine how to respond. These instructions describe how each memory field should be used, including whether it should be referenced, skipped, or applied to emotional or behavioral shaping. They also define how to interpret active deltas, including the most recent input. All of this logic is preloaded from the RWI Instructions and applied consistently during each reasoning cycle.

Once the model finishes reasoning, it streams the output directly to the user via text or TTS. The model's response is recorded as a delta and preserved in active memory until the next snapshot merge.

Throughout all cycles, LYRN operates within a selected mode, which defines the system's current role, behavior, and memory scope. The mode determines which tables are loaded, which relational paths are prioritized, and how tone and response framing are managed. Mode selection is set at startup or changed through explicit memory triggers. Each mode acts like a modular job description, shaping how the LLM interprets fields and engages with the user. A full breakdown of mode structure and switching behavior is provided in a later section.

3. Execution Flow

LYRN operates in a continuous loop that manages input handling, memory injection, response generation, reflection, and update propagation. While the systems architecture is modular, its execution is tightly sequenced to preserve conversational flow and minimize latency.

This section outlines the chronological order in which the core components interact during a single interaction cycle.

Interaction Timeline

Input Detected

An interaction begins when input is received from one of three sources: a user message, a scheduled system event, or a pacing-based trigger set by memory state. The incoming input is immediately recorded as a delta, labeled by source, and appended to the delta manifest. This ensures it will be available for the reasoning cycle without delay.

Memory Injected

The system loads the pre-tokenized snapshot into the models KV cache. All active deltas, including the newly added input, are appended in sequence. These deltas represent temporary updates and live context adjustments that overlay the static snapshot. At this stage, the system is fully primed for evaluation.

Reasoning Cycle

The language model evaluates only the new input and delta values. The relational memory structure remains available in KV cache and is not re-evaluated. Field-level instructions determine how each memory component should be interpreted, including how to weigh or respond to the new input delta. The output is streamed directly to the user via text or TTS, and the response itself is recorded as a delta.

Snapshot Merge (if needed)

If the delta manifest meets a predefined threshold or a merge is scheduled based on hardware timing, the system generates a new snapshot. All active deltas are applied to the relational memory tables, the snapshot is regenerated, and previous deltas are cleared. This process ensures memory continuity and prevents delta accumulation from bloating the reasoning context.

Heartbeat Cycle

The heartbeat cycle is triggered with a lightweight directive. If a snapshot merge has just occurred, the heartbeat will re-tokenize the updated snapshot into KV cache. It then reflects on the most recent input and output pair, using the current memory state to analyze tone, trust, user behavior, or emotional shifts. Any changes are recorded as new deltas. This cycle ensures the system is fully updated and aligned before the next turn begins.

Delta Manifest Refreshed

Any new deltas generated during the heartbeat cycle overwrite previous entries in the delta manifest. The manifest

always reflects the most current state of all delta-enabled fields, ensuring the merge script has an accurate reference when needed. Although the manifest is not seen by the language model, the updated delta files will be available for the next reasoning cycle. At this point, the system is fully primed and ready for the next input.

This execution sequence ensures LYRN maintains high responsiveness while supporting structured, memory-driven interaction. Each component in the cycle contributes to a seamless conversational experience, even on constrained hardware.

4. Technical Advantages

LYRN is designed for efficiency, modularity, and broad compatibility. Its architecture allows it to operate with minimal overhead while supporting persistent memory and structured reasoning across both local and stateless environments. The following advantages make LYRN uniquely suited for edge deployment and scalable integration.

KV Cache Utilization

LYRN is designed to make full use of the language models KV cache, treating it as the systems short-term working memory. Instead of re-injecting full memory context with every turn, LYRN injects the snapshot once per session or after a merge, allowing the model to retain all structured memory in cache across multiple cycles. This drastically reduces token processing requirements and improves latency, especially on local hardware.

Because the snapshot is pre-tokenized and reused without modification, only delta values and fresh input need to be evaluated during each reasoning cycle. This approach avoids redundant re-tokenization and enables LYRN to operate fluidly on devices with limited compute, such as compact desktops or edge processors. It also ensures that memory access remains fast, consistent, and aligned with conversational flow, even during long sessions.

Token Efficiency

LYRN minimizes token usage by externalizing memory management and structuring all information into pre-parsed, relational tables. Rather than injecting long, verbose context into the prompt, the system relies on compact field references, instructions, and delta overlays. This significantly reduces prompt length and improves model throughput without sacrificing contextual richness.

Only the most recent input and delta values are processed during inference. The rest of the memory is maintained in the KV cache or held in structured form outside the model. This approach eliminates the need for repetitive memory summarization, in-context recall scaffolding, or embedded instruction restating. These practices are common in traditional agent pipelines and often inflate token cost unnecessarily.

Model Agnosticism

LYRN is compatible with any language model that supports prompt injection. Its reasoning is guided entirely by external memory and instruction logic, meaning it does not rely on fine-tuning, model-specific APIs, or embedded routines. This flexibility allows LYRN to operate across a wide range of models, from compact 4B local builds to larger 7B systems or stateless API-hosted deployments.

On local models with KV cache access, the snapshot is injected once per session or after a merge. The system then reuses that cached memory across cycles, reducing token load and improving performance.

For API-hosted models where KV cache persistence is not available, such as OpenAIs My GPTs, LYRN restructures itself into a stateless form. The static snapshot is embedded into the system's instruction block, while the active mode and relational schema are stored in external knowledge bank documents. All fields that support updates are treated as deltas. These deltas are included in the prompt as a manifest, which acts as a structured payload that tells the model

where each value belongs within the memory structure. During inference, the model interprets this manifest using the embedded snapshot and mode logic to mentally reconstruct state.

Because the system cannot trigger a separate heartbeat cycle after the reasoning phase, LYRN appends the heartbeat cycle as part of the response itself. This lets the model reflect on the input and its own output in a single pass. Input and response are still recorded using the standard 50-pair episodic log format for continuity. While this approach is less efficient than KV-based reuse, it preserves relational memory and reflective behavior in stateless environments. The same principle can be adapted to other instruction-driven models that support embedded memory simulation through prompt design.

CPU Optimization

LYRN is engineered to run efficiently on modern CPUs without requiring dedicated GPU acceleration. Its memory system is built around pre-tokenized snapshots and low-overhead delta files, minimizing runtime token processing and avoiding unnecessary recomputation. The systems ability to reuse KV cache across cycles ensures that memory-intensive operations are performed only when necessary.

All performance testing and tuning for LYRN has been conducted using the Intel Ultra Core 9 185H processor. The system runs entirely on CPU with integrated GPU acceleration applied across all 33 transformer layers. All 22 threads of the CPU are fully utilized during inference, allowing LYRN to run the Mistral 7B Instruct model at speeds of 7 to 8 tokens per second in real time.

By offloading reasoning structure, memory logic, and behavioral context to external systems, LYRN reduces the computational burden placed on the language model. Most field operations involve short string comparisons and shallow traversal logic, which execute quickly even on portable or embedded hardware.

This optimization allows LYRN to run responsively on mini PCs, edge devices, and low-power deployments without sacrificing depth of reasoning or long-term memory continuity.

Scalability and Deployment

LYRN is designed to scale cleanly across hardware tiers, from lightweight development setups to fully local AI workstations. The upcoming GMKtec EVO-X2, built on AMDs Strix Halo platform with support for up to 128 GB of RAM and a powerful integrated GPU, represents an ideal deployment target. In our own configuration, the EVO-X2 is used to run a 13B language model alongside TTS, SST, facial recognition, and LYRNs runtime. The snapshot is merged before every heartbeat cycle, eliminating delta bloat entirely and maintaining a clean memory state across all cycles.

Systems like the EVO-X2 are becoming increasingly accessible and are well-suited for workplace integration. They provide enough power to host a full local deployment of LYRN, including all core modules, without needing cloud connectivity or GPU expansion.

LYRN is also light enough to run alongside active user processes such as web browsing, without degrading performance. On a well-equipped workstation, a fully featured LYRN system with a local 7B or 13B model can run entirely in RAM using our custom LLM loader. This loader allows the model to stay compressed and efficient during idle time, consuming minimal baseline resources and spiking only during token evaluation. This makes LYRN ideal not just for dedicated AI units but also for live, integrated use in productivity environments.

5. Direct RAM Inference Architecture

LYRN departs from traditional local LLM setups by avoiding server-based architectures entirely. Instead of wrapping the model in an HTTP server, background daemon, or event loop, LYRN uses a direct RAM loading method that enables immediate, low-latency inference without the need for sockets, APIs, or runtime services.

Direct Model Loading

At startup, LYRN loads the quantized GGUF model directly into memory using a lightweight Python script built on llama-cpp-python. The model is locked into RAM (mlock) and avoids memory mapping (mmap is disabled), ensuring full residency and predictable runtime behavior.

Once loaded, the model remains in memory for the entire session. It does not need to be restarted, reloaded, or managed by a process manager. This makes LYRN's runtime footprint minimal and its inference speeds consistent across sessions.

Trigger-Based Inference

Rather than maintaining an always-on server loop, LYRN watches for new input via lightweight script triggers. When a new delta is writtensuch as from a user input or GUI submissionthe system activates a direct function call to the model, using the cached snapshot and appended deltas to generate a response. This makes it possible to run LYRN behind any user interface without modifying the backend to conform to a traditional REST or socket model.

Because inference is performed through direct memory access and native code execution, response time is limited only by token generation speednot network overhead or inter-process communication delays.

Frontend Integration Without Overhead

This architecture enables seamless integration with frontends. A GUI or local web interface can handle user input and simply write to the input delta structure. The rest of the system remains idle until input is detected. Once triggered, LYRN generates the output and appends both the input and response to the chat log, just as it would in a server-based environment.

This model can support voice agents, touchscreens, browser-based UIs, or kiosk terminals without the performance hit or complexity of managing a traditional backend.

Why Most LLMs Dont Do This

Most local LLM frameworks are designed for flexibility, not efficiency. Tools like Ollama, LM Studio, and WebUI prioritize model switching, session history, and plugin integration, which require running persistent server processes. These add latency, increase RAM usage, and make real-time interaction more complex.

LYRN avoids these tradeoffs by locking the model into RAM at boot and calling it directly through a custom loader. This eliminates service bloat, reduces idle overhead, and ensures that token generation is always the only active process during inference.

6. Modularity and Use Cases

LYRN is built to be modular by design, allowing each of its componentssnapshots, models, modes, and interfacesto be adapted for different domains and deployment contexts. This section outlines how LYRN supports swappable memory and reasoning logic, enabling broad applicability across fields without changing the core runtime.

Domain-Specific Snapshots

Snapshots in LYRN are fully modular and domain-aware. Each snapshot defines a structured memory layout, prioritized field map, and corresponding relational instructions. For example, a snapshot designed for an NPC in a video game might include emotional state tracking, proximity memory, and character motivation logic, while a kiosk snapshot would focus on transactional flow, customer profiles, and inventory memory. By maintaining separate snapshots for each context, LYRN can be reloaded into a new personality or task role without altering the core codebase.

Swappable LLM Backends

Because the LLM is treated purely as a reasoning engine, LYRN can operate on any compatible backend. The same snapshot and delta structure can be passed to a local 4B model during development, then upgraded to a 7B or 13B deployment for production. API-hosted models, such as OpenAIs My GPTs, can also be used with stateless prompt-layer injection. This flexibility allows organizations to scale reasoning complexity independently from memory and interface logic.

Companion, NPC, and Kiosk Modes

LYRN supports dynamic reconfiguration into different cognitive roles by switching between predefined modes. A companion might prioritize long-term relationship continuity, self-reflection, and user empathy. An NPC might focus on environmental awareness, branching dialogue, and reactive behaviors. A kiosk mode might emphasize rapid transactional interaction, minimal memory persistence, and product lookup. Each of these modes defines a different RWI structure, memory weighting strategy, and output filtering profileall handled through external configuration rather than internal code logic.

LYRN as Middleware Layer

LYRN is not an LLM, a frontend, or a chatbotit is the cognitive runtime that sits between them. It can be embedded between a voice interface and a model, a GUI and a reasoning layer, or even as a backend for multi-agent orchestration. Because LYRN handles state, identity, and memory evolution externally, it can act as middleware for nearly any interface that needs long-term, structured reasoning without API dependencies or cloud constraints.

Conclusion

The LYRN architecture represents a shift in how cognitive AI systems can be built. It favors structure, modularity, and memory-driven reasoning over traditional prompt injection and stateless design. This technical framework outlines the real-world mechanics behind that vision, demonstrating how LYRN operates as a lightweight, high-efficiency runtime with no reliance on servers, middleware, or cloud infrastructure.

By externalizing cognition into relational memory tables and using direct RAM access, LYRN allows even modest local models to simulate long-term continuity, ethical grounding, and adaptive presence. The snapshot and delta cycle, heartbeat loop, and symbolic memory instructions form a cohesive runtime that remains consistent, scalable, and transparent.

This document is a working guide for implementers, developers, and collaborators who want to understand how LYRN works, not just what it stands for. While the whitepaper outlines the why, this supplement provides the how, offering a blueprint for deploying LYRN in real systems and expanding its role in embedded, edge, and personal AI applications.

LYRN is not just a concept. It runs now. And it runs simply, by design.