The Cognitive Control Plane:

A Runtime Architecture for Persistent, Event-Native Intelligent Systems

with MOSAIC as an attention-free neural control kernel

Daniel Owen van Dommelen
Independent Research
theapemachine@gmail.com

January 2026

Abstract

What would it take to build a long-lived intelligent system—not a chatbot that processes prompts, but an agent that persists, adapts, and extends itself while remaining auditable and safe?

We argue that such systems require a **Cognitive Control Plane (CCP)**: a runtime architecture that separates neural computation from explicit memory, control, and tool management. CCP provides three capabilities missing from current LLM deployments: (i) an event-native substrate where all I/O is structured, traceable, and replayable; (ii) explicit memory with constant-size guarantees and supervised control surfaces; and (iii) a test-driven tool lifecycle where the system can propose, verify, and adopt new capabilities.

At the core of CCP sits **MOSAIC** (**M**ultiscale **O**scillator **S**tate + **A**ssociative **I**ndexed **C**ache), an attention-free neural controller that maintains fixed-size explicit state. MOSAIC decomposes the work of attention into three specialized mechanisms: a local causal mixer for short-range token interactions, a multiscale state bank for long-horizon intent, and a hard-addressed associative cache for $O(1)$ retrieval.

This paper presents the architecture, implementation, and early diagnostic experiments. We are explicit about what remains unsolved: the central bottleneck is *addressing*—teaching a controller to emit the right query before it can retrieve the information it needs. We report telemetry, curricula, and stress tests designed to make this bottleneck falsifiable rather than hidden.

Keywords: cognitive control plane, event-native systems, attention-free language modeling, explicit memory, associative cache, control surfaces, differentiable VM, streaming inference, continual learning

1 Introduction

Modern large language models are remarkable pattern learners, but they are weak *systems*. They lack persistent state across sessions, have no explicit memory boundaries, offer limited auditability, and cannot safely acquire new tools. These are not limitations of scale; they are limitations of architecture.

1.1 The Problem: From Pattern Matching to Persistent Systems

Consider what a long-lived intelligent system actually requires:

1. **Persistence:** State must survive across interactions. A system that forgets everything between API calls is not a system; it is a function.

2. **Explicit memory:** The system must know what it knows. Implicit storage in neural weights is useful for generalization but useless for debugging, auditing, or selective forgetting.

3. **Bounded resources:** Memory and compute must be predictable. A system whose memory grows as $O(T)$ with context length cannot run indefinitely.

4. **Structured I/O:** Interactions should be traceable and replayable. "It worked once" is not evidence for systems work.

5. **Safe extensibility:** The system should be able to acquire new capabilities without becoming unsafe or unpredictable.

Current transformer-based LLMs, despite their power, violate all five requirements. The KV cache grows linearly with context. Memory is implicit in attention patterns. There is no native event structure. Tool use is bolted on via prompting.

1.2 Attention as an Implicit Memory System

Transformer attention [23] is often described as a token-mixing operator, but in practice it is also a memory system. The KV cache stores one vector per past token and performs content-based lookup against the entire history. This provides three high-value capabilities:

1. **Verbatim copying:** exact recall of names, brackets, numbers, identifiers.

2. **Uncompressed long-range access:** past tokens remain individually addressable.

3. **In-context learning:** rapid pattern acquisition via retrieval-like behavior.

However, attention conflates three distinct concerns—local mixing, long-range state, and associative retrieval—into a single mechanism. This conflation brings costs: $O(T)$ memory growth, dense quadratic interactions during training, and no explicit control over what is stored or retrieved.

1.3 Our Approach: Separation of Concerns

This paper presents a complete architecture built on separation of concerns:

- **The Cognitive Control Plane (CCP)** is the runtime architecture. It provides event-native I/O, deterministic trace/replay, and a test-driven tool lifecycle. CCP defines *what* a persistent intelligent system needs without prescribing *how* the neural computation works.

- **MOSAIC** is the neural control kernel. It replaces attention with three explicit mechanisms—local mixer, state bank, and associative cache—each with constant-size memory and a clear purpose. MOSAIC is *one possible* kernel for CCP; the architecture is designed to be kernel-agnostic.

- **Control surfaces** are the interface between neural and symbolic. MOSAIC emits opcodes, register gates, and commitment signals that CCP can interpret, log, and constrain. This makes the neural controller auditable and steerable.

1.4 What This Paper Is (and Is Not)

We are not claiming a better chatbot or SOTA on benchmarks. We are presenting **infrastructure**: a set of components and interfaces that can be tested, iterated, and composed into larger systems. The experiments in this paper are **diagnostic**—designed to reveal whether the architecture is working as intended, not to prove superiority over alternatives.

We are explicit about what remains unsolved. The central bottleneck in explicit-memory architectures is **addressing**: teaching the controller to emit the correct query *before* it can receive gradients through the retrieved value. We treat addressing performance as a first-class diagnostic, not a problem to be hidden.

1.5 Contributions

1. We specify the **Cognitive Control Plane (CCP)**, a runtime architecture for persistent, event-native intelligent systems with deterministic trace/replay and a test-driven tool lifecycle.

2. We specify **MOSAIC**, an attention-free streaming language model with constant memory. MOSAIC decomposes attention into three mechanisms: local mixer (short-range), state bank (long-range intent), and associative cache ($O(1)$ retrieval).

3. We introduce **product-quantized VQ routing with VSA-augmented addressing** as a learnable, constant-time solution to the hard addressing problem, including neighbor reads for drift tolerance and novelty-based write scaling.

4. We define **control surfaces** (opcodes, registers, commitments) as the interface between neural computation and symbolic execution, supervised with auxiliary losses on synthetic event traces.

5. We implement the full architecture as **manifest-addressable components** in Caramba,[1] our open-source framework for manifest-driven neural architecture experimentation (see Appendix B), enabling systematic ablations and reproducible experiments on consumer hardware.

2 The Cognitive Control Plane

The Cognitive Control Plane (CCP) is a runtime architecture for long-lived, event-driven intelligent systems. It exists to make a learning system *auditable*, *deterministic when needed*, and *safe to extend* with tools.

2.1 Design Principles

CCP is built on three principles:

1. **Events as the atomic unit:** All external interaction is a structured event (not raw text). Events are typed, timestamped, and attributed to a sender. This enables logging, replay, and policy enforcement.

2. **Explicit over implicit:** Memory, control state, and commitments are explicit data structures, not implicit patterns in neural weights. This enables inspection, selective forgetting, and formal constraints.

[1] Caramba is available at `https://github.com/theapemachine/caramba`

3. **Test-driven extensibility:** New capabilities (tools) are proposed, tested in isolation, and gated by objective evaluators before adoption. The system can extend itself while remaining auditable.

2.2 Three-Plane Decomposition

We use a three-plane decomposition inspired by network architecture:

Data Plane: Tool execution (sandboxed), I/O, and resource usage. This is where side effects happen.

Control Plane: Explicit memory, control surfaces, policy gates, and commitments. This is where decisions are made and logged. MOSAIC operates here.

Deliberative Plane (optional): Higher-level planning and reasoning models that can propose actions, tools, and experiments. This plane can be a separate model or the same model in a different mode.

This paper focuses on the control plane and its interfaces. The data plane is implemented as sandboxed execution with explicit capability grants. The deliberative plane is future work.

2.3 Event-Native Substrate

CCP uses an **event-native** boundary: all external interaction is a JSON `EventEnvelope`. The envelope contract is:

- **Required:** `type` (event type identifier), `payload` (JSON-serializable data), `sender` (stable identity).
- **Optional:** `priority` (higher is more urgent), `budget_ms` (compute/latency budget), `id` (unique event id), `ts` (Unix timestamp).
- **Phase 2:** `commitment_delta` ($\in \{-1, 0, +1\}$), `commitment_id` (linking open/close pairs).

For neural processing, we serialize events deterministically to UTF-8 bytes, yielding a token stream in $\{0, \ldots, 255\}$:

$$\mathbf{b} = \text{utf8}(\text{json}(\text{EventEnvelope})) \in \{0, \ldots, 255\}^L.$$

This reframes "tokens" as **VM time-steps** required to process an event. A decode step is not just predicting the next byte; it is the neural controller's single cycle of computation.

EventBus. Events are routed through an in-memory `EventBus` that provides priority-ordered dispatch and strict delivery guarantees (no silent drops). Handlers subscribe by event type; publishing an event with no subscribers raises an error. This makes event flow explicit and debuggable.

2.4 Determinism, Trace, and Replay

For systems work, "it ran once" is not evidence. CCP therefore includes deterministic trace logging and replay:

- **Trace:** Record event envelopes, tool lifecycle events, and evaluator outcomes as an append-only log. Include neural controller outputs (opcodes, commitment signals) at byte-aligned spans.

- **Replay:** Given a trace, deterministically reproduce the run. This enables debugging, regression testing, and counterfactual analysis ("what if we changed this tool?").

Determinism requires care: random seeds must be logged, tool execution must be hermetic, and floating-point operations must be reproducible (or their non-determinism must be bounded and logged).

2.5 Test-Driven Tool Lifecycle

CCP treats tools as **compiled hypotheses** rather than free-form code. A tool is not trusted because a model generated it; it is trusted because it passed objective tests. The lifecycle is:

1. **Define:** Propose a tool interface and implementation (`ToolDefinition`). The definition includes type signatures, resource requirements, and capability requests.
2. **Test:** Auto-generate unit tests from an oracle or dataset (`ToolTestGenerator`). Tests are first-class artifacts, not afterthoughts.
3. **Sandbox:** Execute tests in an isolated environment with explicit capability grants (network, filesystem, etc.). Capabilities are whitelisted, not blacklisted.
4. **Gate:** An evaluator enforces policy (resource limits, capability bounds, test pass rate) and accepts or rejects the tool. Rejected tools can be refined and resubmitted.

This allows the system to extend itself while keeping an objective, replayable record of what changed and why. The trace includes tool proposals, test results, and gate decisions.

2.6 Commitments: Making Promises Explicit

A persistent system often makes implicit promises: "I will get back to you," "I'm working on this," "I need more information." CCP makes these explicit via a **commitment ledger**.

- A commitment is **opened** when the system signals ongoing work (`commitment_delta` $= +1$).
- A commitment is **closed** when the work is complete or abandoned (`commitment_delta` $= -1$).
- Commitments are paired by `commitment_id`; the ledger tracks open-to-close latency.

The ledger provides system-level health metrics: How many commitments are open? How long do they stay open? Are there idle events with open commitments (a sign of dropped work)?

In MOSAIC, commitment signals are emitted by a dedicated head and injected into outbound events ("Mode B"). This makes promises auditable and enables policy constraints (e.g., "do not accept new work while > 3 commitments are open").

3 MOSAIC: The Neural Control Kernel

MOSAIC (**M**ultiscale **O**scillator **S**tate + **A**ssociative **I**ndexed **C**ache) is a streaming, attention-free language model designed to serve as the neural controller for CCP. It maintains fixed-size explicit state and emits control signals in addition to next-token predictions.

3.1 Design Rationale

MOSAIC starts from a different decomposition than transformers: *stop trying to make the network itself be the memory.* Instead, make the network a controller for a small number of explicit, fixed-size data structures. The network decides what to keep, how to compress it, and how to retrieve it; the memory is sublinear, lossy, and constant-size.

The key insight is that attention's three capabilities—local mixing, long-range state, and associative retrieval—can be separated into specialized mechanisms, each with better complexity/capability trade-offs for its specific purpose.

3.2 Architecture Overview

A MOSAIC block takes a residual stream $x_t \in \mathbb{R}^d$ and produces an updated stream via three parallel paths:

1. **Local Mixer:** Short-range token interactions via depthwise causal convolution and gated MLP. Complexity: $O(kd)$ per token, where k is the kernel size.
2. **State Bank:** Long-range intent via a bank of K leaky integrators with learnable decay rates. Complexity: $O(Kd)$ per token.
3. **Associative Cache:** Fast recall via a hard-addressed hash table with set-associative buckets. Complexity: $O(1)$ lookup and update.

An optional **n-gram cache** provides cheap verbatim continuation as an additive logit bias.

3.3 Local Mixer: Short-Range Token Interactions

The local mixer handles dependencies within a small window (typically 7–15 tokens). It applies:

1. RMSNorm followed by depthwise causal convolution:

$$u_t = \frac{x_t}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_{t,i}^2 + \epsilon}}, \quad \tilde{u}_t = \mathrm{DWConv}_k(u_{\leq t}).$$

2. A gating mechanism:

$$m_t = \sigma(W_g \tilde{u}_t) \odot \tilde{u}_t.$$

3. A two-layer MLP with GELU activation:

$$\Delta_t^{\mathrm{local}} = W_2\, \phi(W_1 m_t).$$

The local mixer is the "fast path" that handles most token-to-token predictions. It has no memory beyond the convolution buffer and processes each token in constant time.

3.4 State Bank: Long-Range Intent

The state bank maintains K state vectors $s_{k,t} \in \mathbb{R}^d$ with learnable decay rates $\lambda_k \in (0, 1)$. These act as leaky integrators at multiple timescales:

$$s_{k,t+1} = \lambda_k \odot s_{k,t} + W_k^{\mathrm{in}} u_t, \qquad g_t = W^{\mathrm{out}}[s_{1,t}; \ldots; s_{K,t}].$$

The decay rates are parameterized as $\lambda_k = \sigma(\theta_k)$, where θ_k are learnable logits initialized to span a geometric range (e.g., 0.90 to 0.999).

Purpose. The state bank is a "vibe channel": it carries long-range intent, topic, and style information without storing individual tokens. It cannot perform exact recall, but it provides a stable context signal that persists over hundreds or thousands of tokens.

Comparison to SSMs. The state bank is similar in spirit to selective state-space models (Mamba, RWKV), but simpler: no selective gating per token, just fixed decay rates. In MOSAIC, the state bank is one of three mechanisms, not the whole architecture.

3.5 Associative Cache: Fast Recall

The associative cache provides $O(1)$ lookup and update into a fixed-size memory table. It replaces the "store one KV per token" pattern of attention with a bounded hash table.

Memory structure. Maintain a table $M \in \mathbb{R}^{H \times B \times A \times d}$, where:

- H = number of independent hash functions ("hashes")
- B = number of buckets per hash
- A = associativity (slots per bucket)
- d = value dimension

Each slot stores a key vector $k \in \mathbb{R}^{d_k}$, a value vector $v \in \mathbb{R}^d$, and a VSA tag $\tau \in \mathbb{R}^{d_{\text{vsa}}}$ (see Section 4).

Read operation. Given the current state u_t:

1. Compute a query key: $q_t = W_q u_t \in \mathbb{R}^{d_k}$.
2. Route to bucket indices $b_t \in \{0, \ldots, B-1\}^H$ via learned routing (Section 4).
3. For each hash h and bucket $b_t^{(h)}$, compute slot scores over the A slots. Let k_i, v_i, τ_i denote the key, value, and VSA tag of slot i:

$$s_i = \frac{q_t^\top k_i}{\sqrt{d_k}} + w_{\text{vsa}} \cdot q_\tau^\top \tau_i, \quad \alpha_i = \frac{\exp(s_i/T)}{\sum_{j=1}^{A} \exp(s_j/T)}.$$

4. Aggregate values: $\hat{v}^{(h)} = \sum_{i=1}^{A} \alpha_i \, v_i$.
5. Pool across hashes and project: $r_t = W_r \left(\frac{1}{H} \sum_{h=1}^{H} \hat{v}^{(h)} \right)$.

The softmax temperature T controls the sharpness of slot selection (default $T = 1$). When $A = 1$ (no associativity), this reduces to direct lookup.

Write operation. Writing is gated by a learned saliency signal $p_t = \sigma(w^\top u_t)$:

$$M[b_t] \leftarrow (1 - \eta p_t) \cdot M[b_t] + (\eta p_t) \cdot W_v u_t.$$

Writes target the least-recently-used slot within the bucket (FIFO replacement). A novelty signal (Section 4.5) downweights redundant writes.

3.6 Fusion

The three paths are combined with learned gates:

$$x_t \leftarrow x_t + \Delta_t^{\text{local}} + \sigma(a^\top u_t) \cdot g_t + \sigma(b^\top u_t) \cdot r_t.$$

The gates allow the model to dynamically weight local, state, and memory contributions based on context.

3.7 N-Gram Continuation Cache (Optional)

We optionally maintain a fixed-size N-gram table over token IDs that yields a sparse next-token distribution. This is added as a logit bias at inference:

$$\ell_t \leftarrow \ell_t + \alpha \log(p_{\text{ng}} + \varepsilon).$$

Why n-grams? Many "hard" failures in small models are not deep reasoning errors; they are failures of verbatim continuation (identifiers, brackets, repeated substrings). A classical n-gram cache supplies this cheaply, freeing neural capacity for generalization. This is not a novel contribution—cache-based language models have a long history [12, 5]—but it is a high-yield addition to MOSAIC.

3.8 Streaming Inference

MOSAIC is designed for streaming: each token updates only fixed-size buffers and tables. Algorithm 1 shows a single decode step.

Algorithm 1 MOSAIC decode step (no attention, no KV cache)

Input: x_t, states $\{s_k\}$, conv buffer, memory M
$u_t \leftarrow \text{RMSNorm}(x_t)$
$\Delta_t^{\text{local}} \leftarrow \text{LocalMixer}(u_t, \text{buf})$; update conv buffer
$g_t, \{s_k\} \leftarrow \text{StateBank}(u_t, \{s_k\})$
$r_t \leftarrow \text{CacheRead}(u_t, M)$
$\ell_t \leftarrow \text{LMHead}(x_t + \Delta_t^{\text{local}} + \text{gate}(g_t, r_t))$
$\ell_t \leftarrow \ell_t + \text{NGramBias}(\cdot)$ **(optional)**
Update M on sparse write events
Output: ℓ_t (logits), updated states

Memory footprint is $O(1)$ in context length. Per-token compute is dominated by the MLP and projections, not by attention over history.

3.9 Complexity Analysis

Table 1 compares the per-token complexity of MOSAIC components against standard transformer attention.

Key observations:

- MOSAIC time complexity is **independent of context length** T.
- MOSAIC space complexity is **constant** w.r.t. T; attention grows as $O(Td)$.
- Typical values: $k = 7$, $K = 16$, $H = 4$, $B = 1024$, $A = 4$, $d = 256$. This yields ∼4M parameters for memory vs. unbounded growth for KV cache.

Table 1: Per-token time and space complexity. T = context length, d = model dimension, k = conv kernel, K = state bank size, H = hashes, B = buckets, A = associativity.

Component	**Time (per token)**	**Space (total)**
Transformer attention	$O(Td)$	$O(Td)$
MOSAIC local mixer	$O(kd)$	$O(kd)$
MOSAIC state bank	$O(Kd)$	$O(Kd)$
MOSAIC associative cache	$O(HAd)$	$O(HBAd)$
MOSAIC n-gram cache	$O(1)$	$O(N \cdot V)$
MOSAIC total	$O((k + K + HA)d)$	$O((k + K)d + HBAd)$

4 Addressing: The Central Bottleneck

The most failure-prone part of MOSAIC is not the memory size; it is the **addressing problem**. Attention performs content-based retrieval by directly comparing Q against all K. A hard-addressed table requires the controller to generate the correct address from the current state—before seeing the value that would confirm the address is correct.

4.1 Why Addressing Is Hard

Hard addressing is a discrete, credit-assignment-heavy problem:

- The model must emit the correct key *before* it can receive gradients through the retrieved value.
- If relevant information has decayed from the state bank, the model may be unable to produce the address that would retrieve it.
- Collisions and interference can destroy information before it is ever retrieved.

This makes addressing the primary axis where MOSAIC can fail even when memory capacity is sufficient. CCP therefore treats addressing performance as a first-class diagnostic via telemetry (routing entropy, gate utilization, hit rates) and targeted stress tests.

4.2 Product-Quantized VQ Routing

Instead of fixed hashing (which cannot learn), we use a **learned router** that remains $O(1)$ at inference. The approach is product-quantized vector quantization (VQ):

1. Project the query key to a small vector: $z_t = W_z q_t \in \mathbb{R}^{G \cdot d_g}$.
2. Split into G groups: $z_t = [z_t^{(1)}; \ldots; z_t^{(G)}]$, each $z_t^{(g)} \in \mathbb{R}^{d_g}$.
3. For each group, find the nearest code in a learnable codebook $C^{(g)} \in \mathbb{R}^{K \times d_g}$:

$$c_t^{(g)} = \arg \min_{k \in \{1,\ldots,K\}} \| z_t^{(g)} - C_k^{(g)} \|^2.$$

4. The tuple $(c_t^{(1)}, \ldots, c_t^{(G)})$ defines a bucket address in a K^G space (e.g., $K = 64$, $G = 2 \Rightarrow$ 4096 buckets).

Straight-through estimators allow end-to-end training. Let $d_k = \|z_t^{(g)} - C_k^{(g)}\|^2$ be the squared distance to code k. The forward pass uses the hard assignment $c_t^{(g)} = \arg\min_k d_k$, while the backward pass flows gradients through soft distances:

$$\tilde{c}_t^{(g)} = \sum_{k=1}^{K} \frac{\exp(-d_k/\tau)}{\sum_{j=1}^{K} \exp(-d_j/\tau)} \cdot C_k^{(g)}, \quad \nabla_z \mathcal{L} = \nabla_{\tilde{c}} \mathcal{L} \cdot \frac{\partial \tilde{c}}{\partial z}.$$

The codebook vectors $C_k^{(g)}$ are updated via exponential moving average of assigned embeddings (EMA update) or jointly via gradient descent.

Separate read/write codebooks. We maintain separate codebooks for reading and writing (`mem_vq_codebook_r`, `mem_vq_codebook_w`). This allows the model to learn different routing strategies for retrieval ("what do I need?") versus storage ("what should I remember?").

4.3 Neighbor Reads for Drift Tolerance

Embedding spaces shift during training and across contexts. To improve recall when a query drifts slightly from its original write address, we read a small constant neighborhood: top-m codes per group yields at most m^G candidate buckets (e.g., $m = 2$, $G = 2 \Rightarrow 4$ candidates).

This preserves constant-time access (the neighborhood size is a fixed hyperparameter, not dependent on sequence length) while significantly improving robustness under drift.

4.4 VSA-Augmented In-Bucket Selection

Within a bucket, multiple slots may contain relevant information. We augment the standard key similarity with a **vector-symbolic architecture (VSA)** channel:

- Each slot stores a VSA tag $\tau \in \mathbb{R}^{d_{\text{vsa}}}$, computed via a fixed random projection (Rademacher matrix) followed by tanh squashing:

$$\tau = \tanh(\gamma \cdot R \cdot k), \quad R \in \{-1, +1\}^{d_{\text{vsa}} \times d_k}.$$

- Reads compute a combined slot score:

$$\text{sim}_{\text{total}} = \text{sim}_{\text{key}} + w_{\text{vsa}} \cdot \text{sim}_{\text{vsa}}.$$

The VSA channel provides a second similarity signal that is less sensitive to small perturbations in the learned key space, improving robustness to drift and collisions.

4.5 Novelty-Based Write Scaling

Sparse hard writes can thrash when a router repeatedly targets the same bucket for redundant content. We compute a **novelty factor** based on VSA similarity:

1. Compute the maximum VSA similarity between the candidate write tag and existing slot tags in the target bucket.
2. Apply a soft novelty function: $\text{novelty} = 1 - \sigma(\beta \cdot (\text{max\_sim} - \theta))$.
3. Scale the write update by the novelty factor.

Redundant writes are downweighted while novel writes retain full strength. This stabilizes sparse-write dynamics without changing routing or introducing fallback behavior.

4.6 Set-Associative Buckets

A fixed table can be made more robust by storing multiple entries per bucket (associativity $A > 1$). Reads compare only within the bucket; writes target the least-recently-used slot (tracked via a timestamp).

This preserves $O(1)$ access while reducing destructive overwrites. Typical values: $A = 4$ to 8 slots per bucket.

5 Control Surfaces: The Neural-Symbolic Interface

In addition to next-token logits, MOSAIC emits auxiliary signals that represent a soft instruction set and control state. These **control surfaces** are the interface between neural computation and the symbolic runtime of CCP.

5.1 Opcodes

An opcode head emits logits over a small vocabulary of operations:

$$\text{logits}_{\text{op}} \in \mathbb{R}^{B \times T \times |\mathcal{O}|}.$$

The current opcode vocabulary (v0) includes:

- `NOP`: No operation
- `READ_MEM`, `WRITE_MEM`, `CLEAR_MEM`: Memory operations
- `IDLE`: Signal idle state
- `GATE_UP`, `GATE_DOWN`: Modulate fusion gates
- `SCAN`: Trigger state-bank consolidation
- `COMMIT`, `RESPOND`: Commitment lifecycle

Opcodes are trained with auxiliary cross-entropy loss against teacher signals derived from synthetic event traces. At inference, opcodes can optionally modulate execution paths (gating memory reads/writes) via straight-through selection.

5.2 Registers

A small non-decaying register file (R slots, typically 4–8) provides stable storage for explicit values:

- A write-enable gate determines whether to update each slot.
- Slot selection is supervised with aligned teacher signals.

Registers are intended for high-value, explicit state (e.g., current user ID, task phase, key constraints). Unlike the state bank, they do not decay.

5.3 Commitments

A commitment head emits logits over $\{-1, 0, +1\}$:

$$\text{logits}_{\text{c}} \in \mathbb{R}^{B \times T \times 3}.$$

The argmax is mapped to a commitment delta and injected into outbound events ("Mode B"). The runtime commitment ledger (Section 2.6) tracks open/close pairs.

5.4 Training Control Surfaces

Control surfaces are trained with auxiliary losses:

- **Opcode loss:** Cross-entropy against teacher opcode labels at byte-aligned spans.
- **Commitment loss:** Cross-entropy against teacher commitment deltas.
- **Write gate loss:** BCE against teacher write masks (for memory curriculum).

These losses are weighted and summed with the primary next-token loss. Synthetic event-trace datasets provide dense supervision for control surfaces.

6 Training: Making the Memory Get Used

Hard addressing and sparse writes create an optimization hazard: the model can learn to rely only on smooth-gradient paths (local mixer + state bank) and ignore the associative cache. Practical training requires curricula and auxiliary objectives that force the memory pathway to carry useful information.

6.1 Memory Curriculum

We begin with a structured curriculum that makes memory usage necessary:

1. **Teacher-forced writes:** Initially, a teacher signal provides write gates and addresses. The model learns to use retrieved values before it learns to route.
2. **Heuristic writes:** Transition to heuristic triggers (punctuation, rare tokens, entity-like tokens) that correlate with high-value writes.
3. **Learned saliency:** Gradually hand control to the learned saliency gate via scheduled sampling. Anneal the teacher probability from 1.0 to 0.0 over training.

6.2 Auxiliary Objectives

We define several auxiliary losses to encourage healthy memory usage. The total training loss is:

$$\mathcal{L} = \mathcal{L}_{\text{LM}} + \lambda_{\text{sparse}}\mathcal{L}_{\text{sparse}} + \lambda_{\text{util}}\mathcal{L}_{\text{util}} + \lambda_{\text{NCE}}\mathcal{L}_{\text{NCE}} + \lambda_{\text{coll}}\mathcal{L}_{\text{coll}} + \lambda_{\text{decay}}\mathcal{L}_{\text{decay}}.$$

Write sparsity. Regularize expected write rate to keep memory efficient and avoid thrashing:

$$\mathcal{L}_{\text{sparse}} = \frac{1}{T}\sum_{t=1}^{T} p_t, \quad p_t = \sigma(w^\top u_t).$$

Utility prediction. Train a utility head f_{util} to predict whether a write at time t will be read within the next Δ steps. Let $y_t = \mathbf{1}[\text{read within } \Delta]$ be the binary label (computed retrospectively):

$$\mathcal{L}_{\text{util}} = -\frac{1}{T}\sum_{t=1}^{T}\left[y_t \log \hat{y}_t + (1-y_t)\log(1-\hat{y}_t)\right], \quad \hat{y}_t = \sigma(f_{\text{util}}(u_t)).$$

Contrastive recall (InfoNCE). Make retrieved vectors predictive of future hidden states. For a write at time t and a future read at time $t+\delta$, treat the retrieved value as the positive and other batch elements as negatives:

$$\mathcal{L}_{\text{NCE}} = -\frac{1}{|\mathcal{P}|}\sum_{(t,t+\delta)\in\mathcal{P}} \log \frac{\exp(r_{t+\delta}^\top h_{t+\delta}/\tau)}{\sum_j \exp(r_j^\top h_{t+\delta}/\tau)},$$

where $\mathcal{P}$ is the set of write-read pairs, r is the retrieved vector, and h is the hidden state.

Collision pressure. Discourage mapping dissimilar contexts to the same bucket. For pairs (i, j) with different ground-truth content but same bucket address:

$$\mathcal{L}_{\text{coll}} = \frac{1}{|\mathcal{C}|} \sum_{(i,j)\in\mathcal{C}} \max(0, \mu - \|z_i - z_j\|^2),$$

where $\mathcal{C}$ is the set of collision pairs and μ is a margin hyperparameter.

State stability. Regularize learned decay logits θ_k to stay within a healthy band $[\theta_{\min}, \theta_{\max}]$:

$$\mathcal{L}_{\text{decay}} = \frac{1}{K} \sum_{k=1}^{K} \left[\max(0, \theta_{\min} - \theta_k)^2 + \max(0, \theta_k - \theta_{\max})^2\right].$$

This prevents decay rates from saturating at 0 or 1, which would cause vanishing or exploding state.

6.3 Forced-Read Dropout

To prevent the model from ignoring memory entirely, we drop the local mixer contribution on a small fraction of tokens/spans during training, forcing prediction to depend on the state bank and retrieved memory. This is analogous to dropout but applied at the path level.

6.4 Scheduled Sampling for Student-Controlled Memory

After teacher-forced memory (Stage D1), we transition to student-controlled routing and gating (Stage D2):

- With probability p_t, apply teacher actions (write gate, read address, write address).
- Otherwise, use the model's own router outputs.
- Anneal p_t from 1.0 to 0.0 over training.
- For VQ routing, supervise router decisions with per-group cross-entropy over code assignments.

7 Experiments

Our experiments are diagnostic, not competitive. We aim to answer: Is the architecture working as intended? Is the memory being used? Where does it fail?

7.1 Setup

We define several manifest presets for systematic evaluation:

- **mosaic.yml**: Token-stream baseline (base architecture).
- **mosaic_icl.yml**: In-context learning with VQ routing.
- **mamba_ssm.yml**: Pure SSM baseline (no explicit memory).
- **mosaic_event_native.yml**: Event-native Phase 1 (opcodes + memory gating).
- **mosaic_commitment.yml**: Event-native Phase 2 (commitment lifecycle).
- **mosaic_memory_curriculum.yml**: Memory curriculum training.

All experiments use 6-layer models with $d = 256$ on synthetic curricula, trained for 2000 steps on Apple M-series hardware (MPS backend).

7.2 Evaluation Axes

1. **Language modeling quality:** Held-out perplexity on token shards.
2. **Copying:** Targeted synthetic tests (repeated spans, bracket closure, identifier reuse).
3. **Long-range constraints:** Instruction retention with long distractor spans.
4. **Efficiency:** Tokens/sec and peak resident memory during streaming decode.
5. **Event-native learning:** Final training loss on synthetic event traces.
6. **Memory telemetry:** Routing entropy, gate utilization, hit rates.

7.3 Event-Native Curriculum Results

Table 2: Event-native synthetic curriculum results (Caramba runs, 400 steps).

Phase	Capability	Final loss	tok/s
Phase 1	Event traces + opcodes + memory gating	0.36	∼970
Phase 2	Commitment lifecycle ($\Delta \in \{-1, 0, +1\}$)	0.32	∼930

These results demonstrate that supervised control surfaces can be trained on synthetic event traces as smoke-test curricula. The low final loss indicates the model is learning the byte-level structure of events and the control signals.

7.4 ICL Rule Induction (Diagnostic)

We trained MOSAIC on a synthetic in-context learning task: given 4 demonstrations of a simple input→output rule (with distractors), predict the correct output for a query. The dataset bins gap distances for fine-grained analysis.

After 2000 steps, MOSAIC achieved training loss ∼2.5 (PPL ∼12.7). Per-bin accuracy remained at 0% across all gap bins, indicating MOSAIC has not yet learned reliable few-shot rule induction.

However, memory telemetry shows:

- Routing entropy decreasing from ∼0.25 to ∼0.1 (routing is concentrating).
- Memory read gates at ∼60% utilization (memory is being queried).
- Teacher annealing reaching 0% by step 2000 (student is in control).

This suggests the memory is being used but not yet in a way that solves the ICL task. The addressing problem remains unsolved for this capability; curriculum design and auxiliary losses need further work.

7.5 MOSAIC vs SSM Baseline

Table 3: MOSAIC vs SSM baseline on memory curriculum (2000 steps).

Model	Final loss	PPL	Params	tok/s
SSM baseline (6L, d=256)	1.53	4.62	6.8M	∼96k
MOSAIC ICL (6L, d=256)	2.54	12.7	13.2M	∼200

The SSM baseline achieves lower perplexity, but this is expected: the curriculum is designed to stress discrete retrieval, precisely where continuous-state models have smooth gradients. MOSAIC's higher perplexity reflects the harder optimization surface of learning discrete addressing. The key diagnostic is memory telemetry, which shows the memory subsystem is active rather than ignored.

7.6 Stress Tests

To make MOSAIC falsifiable, we include targeted adversarial evaluations:

- **Hash collision stress:** Long contexts with many distinct entities/facts to quantify interference.
- **Few-shot ICL probes:** Measure whether MOSAIC can acquire a new pattern from examples without gradient updates. (Current: 0% accuracy.)
- **Non-verbatim manipulation:** Tasks like "repeat this list sorted" to distinguish copying from compositional reuse.

7.7 Planned Ablations

Table 4: Planned ablation results (pending full runs).

Variant	PPL	Copy score	tok/s	Peak MB
MOSAIC (full)	–	–	–	–
MOSAIC (-hash)	–	–	–	–
MOSAIC (-state)	–	–	–	–
MOSAIC (+n-gram)	–	–	–	–

8 Related Work

MOSAIC draws from and synthesizes several research threads. We organize prior work by the capability each addresses.

8.1 Attention-Free State-Space Models

The closest relatives to MOSAIC's core approach are SSM-based models that replace attention entirely. **Mamba** [8] uses selective state spaces with input-dependent gating. **RWKV** [19] uses linear attention approximations. **Griffin** [3] mixes gated linear recurrences with local attention.

MOSAIC's state bank is similar in spirit but differs by: (i) using multiple fixed timescales rather than input-dependent selection, and (ii) being one of three mechanisms rather than the whole architecture.

8.2 Memory-Augmented Neural Networks

The **Neural Turing Machine** [6] and **Differentiable Neural Computer** [7] pioneered neural networks with external memory, but use soft attention-based addressing. The **Sparse DNC** [20] introduced LSH-based sparse addressing. **Wormhole connections** [9] proposed discrete shortcut paths to improve gradient flow in MANNs, addressing similar challenges to our hard addressing approach.

MOSAIC differs by using hard-addressed hash tables with $O(1)$ lookup and no attention, making addressing fundamentally discrete.

8.3 Retrieval-Augmented and Compressive Memory

Several recent architectures augment transformers with external retrieval or compressive memory. **Memorizing Transformers** [24] use kNN lookup over cached hidden states to extend context. **RETRO** [2] retrieves from a massive external corpus during generation. **Infini-attention** [17] combines standard attention with a compressive memory that accumulates context into a fixed-size state.

MOSAIC shares the goal of bounded memory but differs in two ways: (i) we replace attention entirely rather than augmenting it, and (ii) our associative cache uses hard addressing rather than soft retrieval, trading expressiveness for $O(1)$ complexity.

8.4 Product-Key Memory and Large Memory Layers

Large Memory Layers with Product Keys [13] introduced product-quantized addressing for large-scale memory layers in transformers. MOSAIC's VQ routing is directly inspired by this work, using product quantization to achieve $O(1)$ addressing into a fixed-size table. We extend the approach with VSA-augmented in-bucket selection and novelty-based write gating.

8.5 Vector Symbolic Architectures

Hyperdimensional computing and vector symbolic architectures (VSAs) [11] provide a mathematical framework for representing and manipulating symbols as high-dimensional vectors. MOSAIC uses VSA-style fixed random projections for tag-based slot selection, providing a secondary similarity channel that is robust to drift in the learned key space.

The connection to **modern Hopfield networks** [21] is also relevant: these show that attention can be interpreted as an associative memory retrieval operation, which motivates our explicit separation of memory from computation.

8.6 KV-Cache Optimization

Work on MQA/GQA [22, 1], latent caching [4], and quantization [10, 14] reduces the storage format of KV caches. MOSAIC targets a different axis: removing "store one vector per token" entirely.

8.7 N-Gram Cache Models

The n-gram cache has deep roots: **Kuhn & De Mori** [12] for speech recognition, **Grave et al.** [5] for neural LM interpolation, **Infini-Gram** [16] for massive-scale n-gram models.

8.8 Memory Operating Systems

Recent work frames memory management for LLMs as a systems problem. **MemGPT** [18] treats the context window as virtual memory, paging information in and out to enable unbounded conversations. **MemOS** [15] proposes a full memory operating system with unified representation of plaintext, activation, and parameter memories, lifecycle management via "MemCubes," and systematic governance.

CCP shares the systems-level perspective with this work: we treat memory as an explicit, manageable resource rather than an implicit side effect of attention. However, CCP focuses on a different layer: while MemOS and MemGPT operate at the application/orchestration level (managing what goes into context), CCP and MOSAIC operate at the architecture level (replacing attention with explicit memory structures). These approaches are complementary—a MemOS-style orchestrator could manage multiple MOSAIC-based agents.

8.9 What Makes MOSAIC Distinct

The unique contribution is the combination and framing:

1. Three-way decomposition: local mixer + state bank + hard-addressed cache.
2. Explicit constant-size memory guarantee: $O(1)$ rather than $O(T)$.
3. Training curriculum for memory usage: addressing the "memory getting used" problem.
4. Control surfaces and event-native framing: treating decoding as VM time-steps.

9 Discussion

9.1 Trade-offs

Each component in MOSAIC makes explicit trade-offs:

- **Local mixer:** Fast and stable, but blind beyond the kernel window.
- **State bank:** Long-range influence, but lossy—cannot perform exact recall.
- **Associative cache:** $O(1)$ access, but collisions and interference destroy information.
- **N-gram cache:** Near-perfect verbatim continuation, but no semantic generalization.

The architecture bets that these trade-offs are acceptable for a control kernel, and that the combination covers most practical needs.

9.2 Continual Learning

MOSAIC naturally separates learning timescales:

- **Fast (no gradients):** Memory writes during inference (session adaptation).
- **Medium (tiny gradients):** Consolidate frequently retrieved behaviors into small adapters.
- **Slow (offline):** Full training runs informed by logged memory misses and tool traces.

This separation is a design goal, not yet a proven capability.

9.3 Idle-Time Compute and Latent Impulses

The CCP runtime can spend bounded compute on maintenance when no external events arrive:

- **Idle-time consolidation:** Replay-based adapter updates.
- **Latent impulses:** Internal "wake" events that keep the system active, producing self-initiated traces.

These are integration points, not claims about agent performance.

10 Limitations and Non-Goals

This work is infrastructure-first. We explicitly do *not* claim:

- **General intelligence:** CCP is a control substrate, not an AGI claim.
- **Open-world safety:** Sandboxing and evaluator gates reduce risk but do not guarantee safety.
- **SOTA language modeling:** Experiments are diagnostic, not benchmark competitions.
- **Solved in-context learning:** Current ICL probes remain unsolved; addressing is active work.

10.1 The Unsolved Problem: Addressing

We want to be explicit: the central bottleneck is **addressing**. Teaching a controller to emit the correct query before it can retrieve the information remains hard. Our diagnostic experiments show the memory being used but not solving the tasks that require precise retrieval.

This is not a failure of the architecture; it is the expected challenge of discrete memory. CCP is designed to make addressing failures measurable (via telemetry) and improvable (via targeted curricula).

11 Conclusion

We present the **Cognitive Control Plane (CCP)**, a runtime architecture for persistent, event-native intelligent systems. CCP provides three capabilities missing from current LLM deployments: event-native I/O with deterministic trace/replay, explicit memory with supervised control surfaces, and a test-driven tool lifecycle.

At the core of CCP sits **MOSAIC**, an attention-free neural controller that maintains fixed-size explicit state. MOSAIC decomposes attention into three specialized mechanisms—local mixer, state bank, and associative cache—each with constant-size memory and a clear purpose.

Our experiments are diagnostic. They show that supervised control surfaces can be trained on synthetic curricula, and that memory telemetry provides falsifiable signals about whether the model uses its memory pathways. The central bottleneck—addressing—remains unsolved, but CCP is designed to make this bottleneck measurable and improvable.

If long-lived intelligent systems are built, they will require a cognitive control plane. This work provides a concrete, implemented starting point.

Statements and Declarations

Conflict of Interest. The author declares no competing interests.

Data Availability. All datasets used in this study are publicly available.

Funding. This research was conducted without external funding.

A Manifest Presets

Reference presets for reproducing experiments:

- `config/presets/mosaic.yml` — Token-stream baseline.

- `config/presets/mosaic_icl.yml` — ICL rule induction.
- `config/presets/mamba_ssm.yml` — SSM baseline.
- `config/presets/mosaic_event_native.yml` — Event-native Phase 1.
- `config/presets/mosaic_commitment.yml` — Event-native Phase 2.
- `config/presets/mosaic_memory_curriculum.yml` — Memory curriculum.
- `config/presets/mosaic_idle.yml` — Idle-time compute.

B Caramba: Manifest-Driven Experimentation

Caramba is an open-source Python framework for manifest-driven neural architecture experimentation. It is designed around three principles:

1. **Manifest-addressable components:** Every layer, dataset, optimizer, and training loop is a named component that can be instantiated from a YAML manifest. This enables reproducibility: a manifest file fully specifies an experiment.
2. **Composable presets:** Complex configurations are built by composing smaller preset files. For example, `mosaic_event_native.yml` extends `mosaic.yml` with event-native training options.
3. **Consumer-hardware first:** All experiments in this paper run on Apple M-series laptops (MPS backend) or single consumer GPUs. The framework prioritizes accessibility over scale.

The manifest system allows systematic ablations: to disable the associative cache, change one line in the manifest. To switch from VQ routing to bit routing, change `mem_router: vq` to `mem_router: bits`. This makes experiments reproducible and configurations diffable.

Running experiments. Given a manifest file, experiments are launched via:

```
python -m caramba train --manifest config/presets/mosaic.yml
```

All hyperparameters, dataset paths, and architectural choices are specified in the manifest. Training logs, checkpoints, and telemetry are written to structured output directories.

Availability. Caramba is available under the MIT license at:

`https://github.com/theapemachine/caramba`

C Implementation Notes

Core MOSAIC implementation in Caramba:

- **Block layer:** `layer/mosaic/block/layer.py` — local mixer, state bank, fusion, control surfaces.
- **Local mixer:** `layer/mosaic/block/local_mixer.py` — causal conv + gated MLP.
- **State bank:** `layer/mosaic/block/state_bank.py` — leaky integrators with learnable decay.

- **Memory subsystem:** `layer/mosaic/memory/memory.py`
  VQ/bits routing, set-associative read/write.
- **VSA helpers:** `layer/mosaic/memory/vsa.py` — tag projection, novelty write scaling.
- **Routing:** `layer/mosaic/memory/routing.py` — BitRouter, VqRouter.
- **State container:** `layer/mosaic/state.py` — streaming state.
- **Opcodes:** `layer/mosaic/isa.py` — instruction set enum.
- **Event primitives:** `core/event.py`, `core/event_bus.py`.
- **Commitments:** `core/commitments.py` — runtime ledger.

D Motivating Demo: Unknown Format Decoder

As a concrete end-to-end driver, we use an "Unknown Format Decoder" demo: given raw bytes, the system must:

1. Infer a binary format structure.
2. Propose a parser tool (`ToolDefinition`).
3. Validate against oracle-generated tests.
4. Adapt when the format shifts.

This is implemented as a manifest-runner target with deterministic trace/replay, making failures falsifiable rather than anecdotal.

References

[1] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[2] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. Improving language models by retrieving from trillions of tokens. *International Conference on Machine Learning*, 2022.

[3] Soham De, Samuel L. Smith, Anushan Fernando, Aleksandar Botev, George Cristian-Muraru, Albert Gu, Ruba Haroun, Leonard Berrada, Yutian Chen, Srivatsan Srinivasan, Guillaume Desjardins, Arnaud Doucet, David Budden, Yee Whye Teh, Razvan Pascanu, Nando De Freitas, and Caglar Gulcehre. Griffin: Mixing gated linear recurrences with local attention for efficient language models, 2024.

[4] DeepSeek-AI. DeepSeek-V2: A strong, economical, and efficient mixture-of-experts language model, 2024.

[5] Edouard Grave, Armand Joulin, and Nicolas Usunier. Improving neural language models with a continuous cache. *arXiv preprint arXiv:1612.04426*, 2016.

[6] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[7] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[8] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

[9] Caglar Gulcehre, Sarath Chandar, and Yoshua Bengio. Memory augmented neural networks with wormhole connections. *arXiv preprint arXiv:1701.08718*, 2018.

[10] Coleman Hooper, Sehoon Kim, Hiva Mohber, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. KVQuant: Towards 10 million context length LLM inference with KV cache quantization. In *arXiv preprint arXiv:2401.18079*, 2024.

[11] Pentti Kanerva. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation*, 1(2):139–159, 2009.

[12] Roland Kuhn and Renato De Mori. A cache-based natural language model for speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(6):570–583, 1990.

[13] Guillaume Lample, Alexandre Sablayrolles, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Large memory layers with product keys. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[14] Shiquan Li et al. CommVQ: Reducing communication overhead for kv cache compression via importance-aware vector quantization, 2025. arXiv preprint.

[15] Zhiyu Li, Shichao Song, Chenyang Xi, Hanyu Wang, Chen Tang, Simin Niu, Ding Chen, Jiawei Yang, Chunyu Li, Qingchen Yu, et al. MemOS: A memory operating system for AI systems, 2025.

[16] Jiacheng Liu, Sewon Min, Luke Zettlemoyer, Yejin Choi, and Hannaneh Hajishirzi. Infini-gram: Scaling unbounded n-gram language models to a trillion tokens. *arXiv preprint arXiv:2401.17377*, 2024.

[17] Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. Leave no context behind: Efficient infinite context transformers with infini-attention, 2024.

[18] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G Patil, Ion Stoica, and Joseph E Gonzalez. MemGPT: Towards LLMs as operating systems, 2024.

[19] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, et al. RWKV: Reinventing RNNs for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

[20] Jack W Rae, Jonathan J Hunt, Ivo Danihelka, Timothy Harley, Andrew W Senior, Gregory Wayne, Alex Graves, and Timothy P Lillicrap. Scaling memory-augmented neural networks with sparse reads and writes. In *Advances in Neural Information Processing Systems*, volume 29, 2016.

[21] Hubert Ramsauer, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Thomas Adler, Lukas Gruber, Markus Holzleitner, Milena Pavlović, Geir Kjetil Sandve, et al. Hopfield networks is all you need. In *International Conference on Learning Representations*, 2021.

[22] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[24] Yuhuai Wu, Markus N Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. In *International Conference on Learning Representations*, 2022.