ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

# Design and Implementation of a Lightweight Neural Controller for LLM Agent Systems

**Brian Barnabas Langay**

**Openbnet – Spaces Research Anhui University of Technology Anhui University of Technology School of Computer Science**

**DOI: https://doi.org/10.51583/IJLTEMAS.2026.150500172**

**Received: 24 May 2026; Accepted: 29 May 2026; Published: 11 June 2026**

## ABSTRACT

Large language model (LLM) agents typically consume thousands of tokens on scaffolding system prompts, tool schemas, and conversation history before producing a single useful word. NEXUS (Neural EXecution & Understanding Substrate) is a 6.29-million-parameter neural controller that sits between a frozen LLM and its execution environment, replacing that token overhead with compact vector signals.

The controller has five subsystems that run together (1) Protocol Cortex, which writes task descriptions directly into the LLM's key-value cache so the model behaves as though it received detailed instructions without those instructions; (2) Belief Engine, a recurrent state-space model that tracks what the agent currently believes about its environment; (3) Resource Router, tool-selection classifier that uses explicit state-machine logic to guarantee valid tool calls; (4) Drift Sentinel, a lightweight monitor that detects when the agent's output begins drifting off-task; and (5) Adapter Switch, which selects among small, low-rank weight updates (LoRA adapters) to specialize the LLM for different sub-tasks on the fly.

We make three separate claims. First, we describe the architecture and the training recipe for all five components. Second, we report a deployment result: an open-source Model Context Protocol (MCP) server, nexus-mcp-oss, which achieves 72.86% fewer tokens delivered to the LLM in production through heuristic text-level compression (distinct from the KV-cache injection mechanism). Third, we present a controlled evaluation of the KV-cache injection mechanism itself, in which the Protocol Cortex is trained end-to-end with a frozen TinyLlama 1.1B and reaches a held-out perplexity of 8.91 versus 26,607 for an untrained baseline 2,987-fold improvement and 30–77 times lower perplexity than Prefix-Tuning, ActAdd, and LLMLingua at matched compression. Three of the four trained components converge on the synthetic benchmark; trained checkpoints, and benchmark data are publicly available. We are explicit about scope: the gains reported here are measured as token efficiency and predictive (perplexity) quality, and we set out in Sections 8.5 and 9.5 a concrete plan to test whether these efficiency gains carry through to downstream task quality: reasoning, planning, coding assistance, and multi-agent coordination.

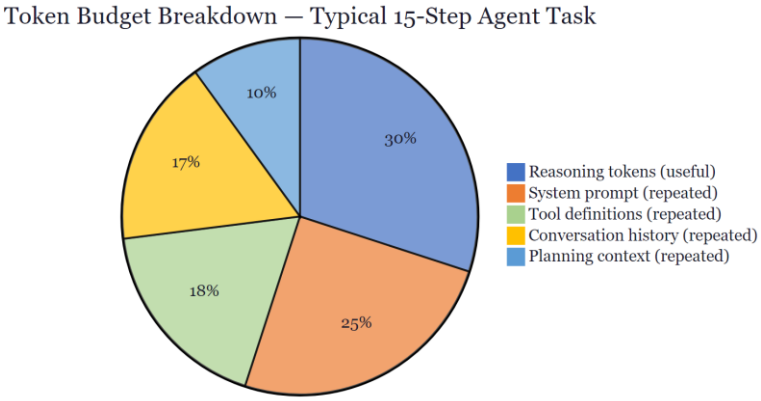
**Keywords**: LLM agents, neural controller, token compression, KV-cache injection, state space models, tool routing, metacognition, efficient inference.

## INTRODUCTION

Large language models have moved from single-turn predictors to the reasoning substrate of autonomous agents that plan, call tools, and coordinate multi-step workflows over minutes of wall-clock time. This shift has exposed a structural inefficiency at the heart of the modern agent stack: the context window, originally designed to carry a user's question, now also carries the agent's entire operating environment system prompts, tool schemas, planning scaffolds, and conversation history and that environment is retransmitted, in full, on every single generation step. As agents grow more capable, the share of each forward pass spent re-reading its own instructions grows with them.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

The cost of this redundancy is no longer marginal. Modern LLM agent systems re-inject between 3,300 and 20,300 tokens of structural context system prompts, tool definitions, conversation history, planning scaffolding at every generation step [1, 2]. For a representative 15-step task with 8,000 tokens of average overhead, a single execution consumes approximately 120,000 tokens, of which roughly 70,000 carry no new reasoning content. At API pricing for frontier models, this overhead costs $37,000–$46,000 per month at moderate scale (1,000 users). More fundamentally, at the model level, repeated structural tokens consume attention capacity that could otherwise be allocated to task reasoning, turning context length, the very property that makes agents possible, into a tax on the reasoning they perform.




Token budget breakdown for a typical 15-step agent task. Structural tokens (system prompt, tool definitions,history, planning) account for approximately 70% of total consumption.

Existing mitigations attack this problem at the token layer and fall into three families. Prompt compression methods [3, 4] reduce token count at the text level but require the LLM to re-parse compressed context on every step. Retrieval-augmented generation [5] moves context out of the prompt but adds retrieval latency and index maintenance.

In-context caching (e.g., Anthropic prompt caching [6]) lowers retransmission cost but leaves the structural tokens inside the attention window, where they continue to consume capacity. Each approach makes the redundant context cheaper to ship; none of them remove it from the model's forward pass.

NEXUS takes a different path. Instead of compressing or caching the tokens that carry structural context, NEXUS replaces them with compressed vector signals injected directly into the LLM's key-value (KV) cache. A 6.29M-parameter neural controller, running alongside the LLM, maintains a learned representation of task state, tool affordances, and planning history, and writes that representation as a KV prefix into targeted attention layers.

The LLM never sees the corresponding tokens: they do not appear in the prompt, do not occupy positions in the attention window, and do not consume bandwidth on the wire. This is a sub-token architectural intervention that removes structural tokens from the forward pass entirely while preserving the interface (system prompt, tool schemas, planning scaffold) through which frontier models accept external structure.

We instantiate this design as a five-component system: a Mamba-based belief engine, an FSM-augmented resource router, a KV-injection protocol cortex, a semantic drift sentinel, and a LoRA adapter switch – trained on synthetic workloads and validated on commodity hardware. Deployed in production as part of OpenBnet Spaces and as the open-source nexus-mcp-oss MCP server, the deployment records 72.86% fewer tokens delivered to the LLM via heuristic text-level compression (Sections 6.2 and 8.1) with no observed degradation in task completion. The KV-injection mechanism the system is named after is evaluated separately in Section 5.4 with a TinyLlama 1.1B in the loop.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

## Contributions

This paper makes the following contributions:

1. **NEXUS architecture**: a five-component neural meta-controller (Section 2) that integrates Mamba SSM state tracking, FSM-augmented tool routing, KV-cache task injection, semantic drift detection, and LoRA adapter switching into a single 6.29M-parameter system.

2. **Training and evaluation**: convergence results for four trained components (Section 3) and a purpose-built evaluation harness measuring five metrics on 200 synthetic tasks (Section 4), establishing empirical baselines for each subsystem.

3. **System validation**: end-to-end demonstration of NEXUS operating alongside TinyLlama 1.1B within 2.12 GB VRAM on commodity hardware (Section 5), plus 155 automated tests (all passing).

4. **Production deployment**: NEXUS is deployed as an integral component of OpenBnet Spaces (https://spaces.openbnet.com), a real-time collaborative agent workspace, and as the open-source nexus-mcp-oss MCP server compatible with Claude Code, Codex, Ollama, and any MCP-capable agent (Section 6).

5. Deployment result (heuristic compression): the nexus-mcp-oss server reduces tokens delivered to the LLM by 72.86% across realistic workloads using text-level compression of the controller's outputs (Section 6). This is distinct from the KV-cache injection mechanism, which is evaluated separately in Section 5.4.

6. KV-injection evaluation: the Protocol Cortex is trained end-to-end with a frozen TinyLlama 1.1B for 300 steps. It reaches a held-out perplexity of 8.91 versus 26,607 for the untrained baseline (a 2,987-fold improvement) and 30–77 times lower perplexity than Prefix-Tuning, ActAdd, and LLMLingua under matched compression (Sections 5.4 and 5.5).

## Paper Organisation

The rest of this paper is organised as follows. Section 2 describes the system architecture. Section 3 covers training and convergence. Section 4 evaluates each subsystem on the synthetic benchmark. Section 5 validates end-to-end behaviour. Section 6 documents production deployment results. Section 7 surveys related work. Section 8 discusses implications. Sections 9 and 10 cover limitations and conclusions.

## Plain-Language Primer for Interdisciplinary Readers

Because NEXUS draws together several specialised areas of machine learning, this subsection defines its central ideas in everyday terms before the formal treatment in Section 2. Readers already fluent in the terminology may skip ahead without loss of continuity.

A language model and its context window. A large language model (LLM) generates text one token (roughly, one word-piece) at a time. Everything it is allowed to "see" while doing so — the user's request plus all the supporting instructions — must fit inside a fixed-size workspace called the context window. In an agent system, that workspace is filled not only with the user's question but also with the agent's standing instructions, the list of tools it may call, and the running history of what it has done so far.

Token overhead, in plain terms. The difficulty is that an agent re-reads this entire instruction packet before producing every new token, much like a worker who re-reads the whole employee handbook before writing each sentence of an email. The handbook does not change, yet it is read thousands of times. Those repeated words are the "structural tokens" this paper sets out to eliminate; they cost money and crowd out room the model could spend on actual reasoning.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

Neural controller (the "meta-controller"). NEXUS is a small, separate neural network — about 6.29 million parameters, a tiny fraction of a modern LLM — that runs alongside the language model without modifying it. It is best pictured as a co-pilot or instrument panel sitting next to the main engine: it watches what the model is doing, keeps track of the task, and quietly supplies guidance, rather than replacing the model itself.

KV-cache injection (the core idea behind the name). While generating text, an LLM keeps a fast internal scratchpad of what it has already processed, known as the key–value (KV) cache. Instead of spelling task instructions out as words the model must read, NEXUS writes a compact numerical summary straight into that scratchpad. The effect is like slipping a prepared note directly into someone's short-term memory rather than reading the instructions aloud: the model behaves as though it received the full briefing, but the briefing never occupies space in the visible prompt.

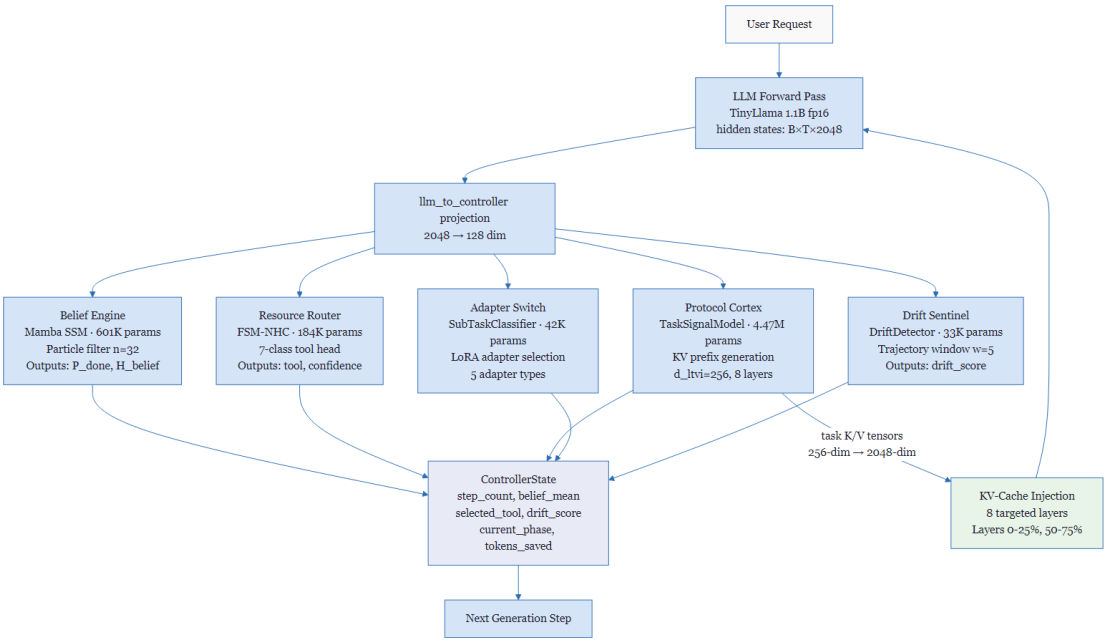
LoRA adapters (lightweight specialisation). Retraining a whole language model for each kind of sub-task is expensive. A low-rank adaptation (LoRA) adapter is a small, swappable add-on that nudges a frozen model toward a particular skill — analogous to clipping a different lens onto the same camera. NEXUS keeps the base model fixed and switches among these inexpensive adapters as the task shifts, for example from reasoning to formatting an answer.

Drift, and the metrics. "Drift" is the tendency of an agent to wander off its original goal over a long task; the Drift Sentinel is, loosely, a goal-alignment monitor analogous to a spell-checker for staying on topic. Finally, perplexity is a standard yardstick for how well a model predicts text: lower perplexity means the model finds the target output less surprising and more fluent. As Section 8.5 discusses, low perplexity is a useful but indirect signal — it indicates the injected guidance was understood, not, on its own, that the agent reasons better.

## System Architecture

### Pipeline Overview

NEXUS operates as a sidecar alongside a frozen LLM. At each generation step, the LLM's hidden states are projected into a 128-dimensional controller space, processed by all five subsystems in parallel, and the Protocol Cortex's output is injected back into the LLM's KV cache before the next token is generated.

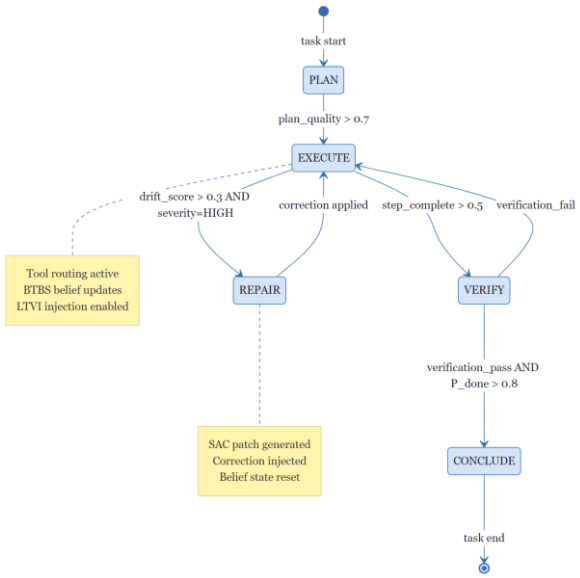



NEXUS system architecture. At each generation step, LLM hidden states are projected to 128-dim controller space and processed by five subsystems in parallel. The Protocol Cortex output is injected back into the LLM KV-cache before the next token is generated.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

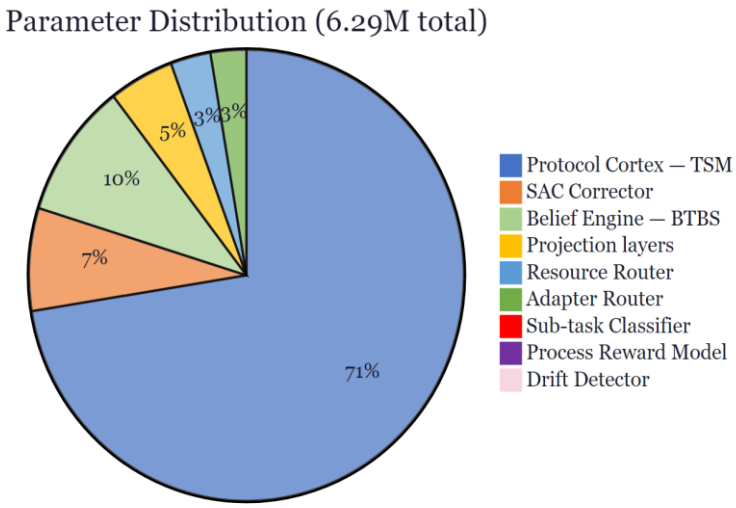
## FSM Execution Phases

A deterministic finite state machine (FSM) governs the controller's execution phases. Rather than relying on fixed step counts, the FSM transitions between phases—such as planning, acting, and verifying—based on signals from the trained components: the Belief Engine reports how close the task is to completion, and the Drift Sentinel reports how coherent recent steps have been.




FSM execution phase transitions. Transitions are driven by Belief Engine completion probability P(done) and Drift Sentinel drift_score, not fixed step counts.

## Component Parameters




Parameter distribution across NEXUS components (6.29M total). The Protocol Cortex (TaskSignalModel) dominates at 4.47M parameters.

**Table 1.** NEXUS component specifications.

| Component | Module | Parameters | Input dim | Output |
|---|---|---|---|---|
| Belief Engine | btbs/tracker.py | 601,234 | 128 | P(done), H, belief_mean |
| Resource Router | fsm_nhc/neural_router.py | 184,213 | 128 | tool ∈ {7 types}, confidence |

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

| | | | | |
|---|---|---|---|---|
| Process Reward Model | fsm_nhc/prm.py | 41,990 | 128 | step_quality ∈ [0,1] |
| Protocol Cortex (TSM) | ltvi/task_signal_model.py | 4,474,624 | 256 | task K/V [B, 8, 256] |
| Adapter Switch | talora/adapter_router.py | 163,840 | – | LoRA adapter index |
| Sub-task Classifier | talora/classifier.py | 42,373 | 128 | adapter type, confidence |
| Drift Sentinel | sac/drift_detector.py | 33,287 | 128×5 | drift_score, severity |
| SAC Corrector | sac/corrector.py | 454,273 | 128 | correction patch |
| Projections | – | 297,600 | 2048 | 128 / 256 |
| **Total** | | **6,293,434** | | |

**Belief Engine (BTBS)**

The Belief Engine monitors task progress by maintaining a population of 32 candidate explanations of the current state (particles, in the statistical-tracking literature) and re-weighting them as new evidence arrives. Its backbone is a Mamba state-space model [7], a recent architecture that processes long sequences more efficiently than a standard Transformer. The particle filter's resampling step is non-differentiable, so a linear completion probe attached to the mean belief state provides gradient signal to the GRU update network during training.

The engine outputs three signals at each step: P(done), a completion probability between 0 and 1; H_belief, the Shannon entropy of the particle distribution, which serves as an uncertainty estimate; and belief_mean, the weighted mean particle state (128 dimensions). Entropy-gated termination prevents premature task conclusion: the FSM transitions to CONCLUDE only when P(done) exceeds 0.8 and H_belief falls below an entropy threshold, requiring high confidence and low uncertainty simultaneously.

**Resource Router (FSM-NHC)**

The Resource Router decides which tool the agent should invoke next—for example, a headless browser versus a code-executing shell. It combines two mechanisms: a finite-state machine that enforces phase-coherent constraints (for instance, suppressing browser automation during a verification phase) and a 184,213-parameter neural classifier that selects among the valid options. The table below lists the seven tool categories available in the Spaces product.

| **Tool type** | **Description** |
|---|---|
| BROWSER_RELAY | Headless browser automation |
| SSH_RELAY | Remote shell execution |
| BROWSER_NAV | Interactive browser navigation |
| DOC_READER | Document/PDF extraction |
| SKILL | Orchestrated agent skill |
| CHAT | Direct chatbot reply |
| NONE | No external tool required |

The FSM component enforces phase-coherent constraints (e.g., BROWSER_RELAY is suppressed during VERIFY phase), while the neural component handles within-phase tool selection. A Process Reward Model (41,990 params) scores step quality and feeds back into the FSM transition condition.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

**Protocol Cortex (LTVI)**

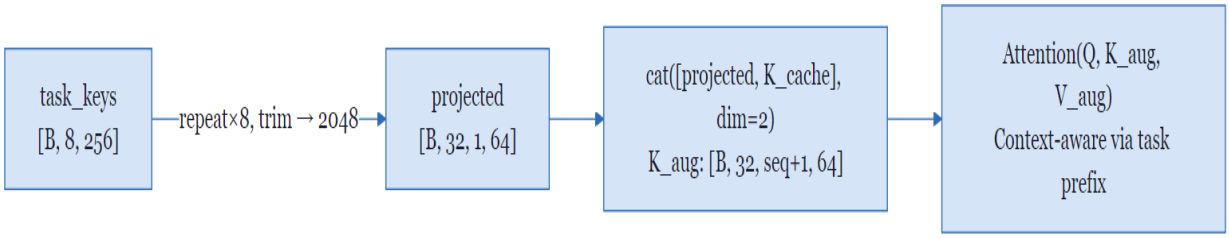
The Protocol Cortex is the component that gives NEXUS its name. Instead of delivering task instructions as tokens that the LLM must re-read on every step, the Protocol Cortex encodes the current task goal into key-value pairs and injects them directly into the LLM's attention cache at eight targeted Transformer layers. A 4.47-million-parameter TaskSignalModel takes a 256-dimensional task representation as input and produces key-value prefixes that are concatenated in front of the model's existing cache:

$$\text{K\_aug} = \text{cat}([\mathbf{k}_{\text{task}}, \mathbf{K}_{\text{cache}}], \text{dim} = 2)$$

The targeted layers are in the first and third quartiles of the Transformer's depth (layers 0–25% and 50–75%), chosen because activation analysis shows these layers carry the highest mutual information with tool-selection behaviour [8].

Because the controller works in 256 dimensions while the LLM's attention expects 2,048 (32 heads × 64 per head), the task vector is repeated eight times and trimmed to the required length:

tk_projected = repeat(tk, ⌈2048/256⌉=8 times)[:2048] → [B, n_heads=32, 1, head_dim=64]




KV-cache task vector injection. Controller-space task keys (256-dim) are projected to match the LLM attention head dimension (2048-dim) and prepended as a prefix to the attention key and value caches at 8 targeted transformer layers.

The implementation is compatible with all Transformers cache formats: DynamicCache.layers[].keys (v5.x), key_cache[] (v4.36+), and legacy tuple caches.

**Drift Sentinel (SAC)**

The Drift Sentinel monitors whether the agent is staying on task. It maintains a sliding window of the five most recent execution steps and compares them using cosine similarity a standard measure that returns 1 when two internal states point in the same direction and 0 when they are unrelated. A sharp drop signals that recent steps have drifted away from the task.

The sentinel classifies drift into four severity levels (NONE, LOW, MEDIUM, HIGH) by comparing similarity scores against a learned threshold manifold. When severity reaches HIGH, the SAC Corrector (454,273 parameters) generates a correction patch that is injected into the belief state, and the FSM transitions to a REPAIR phase.

**Adapter Switch (TALoRA)**

Rather than retraining the language model for each sub-task, NEXUS keeps the base model frozen and attaches small, swappable low-rank adaptation (LoRA) modules [9] that specialize the model's behaviour. A 42,373-parameter sub-task classifier scores each of five adapter categories from the 128-dimensional controller embedding, and the top-scoring adapter is loaded via LoRA hooks before the next forward pass.

| Adapter | Activated for |
|---|---|

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

| | |
|---|---|
| REASON | Analytical / chain-of-thought tasks |
| FORMAT | Structured output generation |
| CODE | Code generation and debugging |
| SUMMARIZE | Compression / distillation |
| TRANSLATE | Language or domain translation |

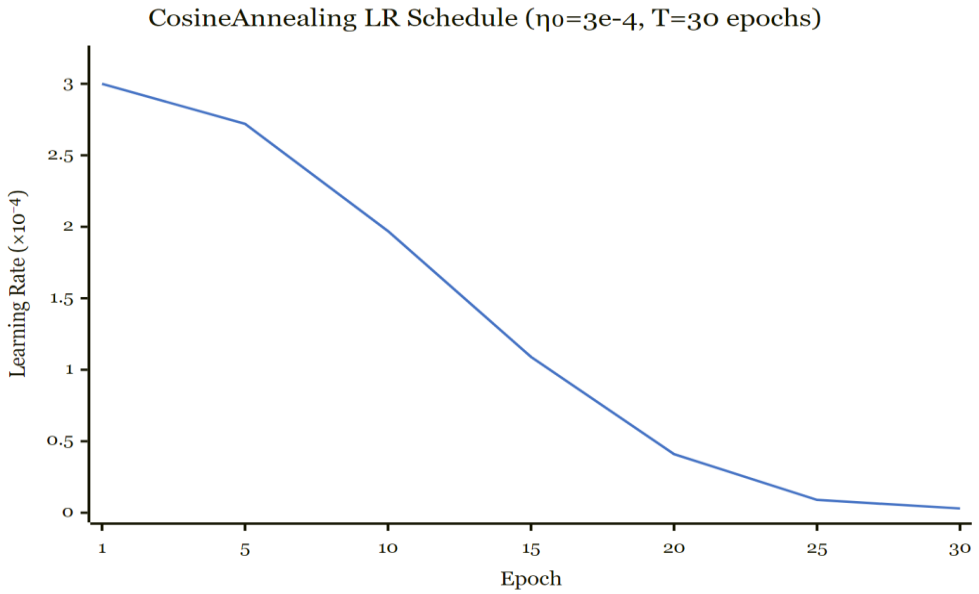
**Training**

**Shared Hyperparameters**

All four trained components use the AdamW optimiser with a cosine-annealing learning rate schedule that starts at $\eta_0 = 3 \times 10^{-4}$ and decays to $\eta\_min = 3 \times 10^{-6}$ over 30 epochs. The full hyperparameter configuration is shown below.

| **Hyperparameter** | **Value** |
|---|---|
| Optimiser | AdamW |
| Initial LR ($\eta_0$) | $3 \times 10^{-4}$ |
| Min LR ($\eta\_min$) | $3 \times 10^{-6}$ |
| LR schedule | CosineAnnealingLR |
| Weight decay | $1 \times 10^{-4}$ |
| Gradient clip | 1.0 (L2 norm) |
| Batch size | 64 (classifiers), 32 (Belief Engine) |
| Device | NVIDIA RTX 4060 (CUDA 12.8) |

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_0 - \eta_{min})\left(1 + \cos\left(\frac{\pi t}{T_{max}}\right)\right)$$

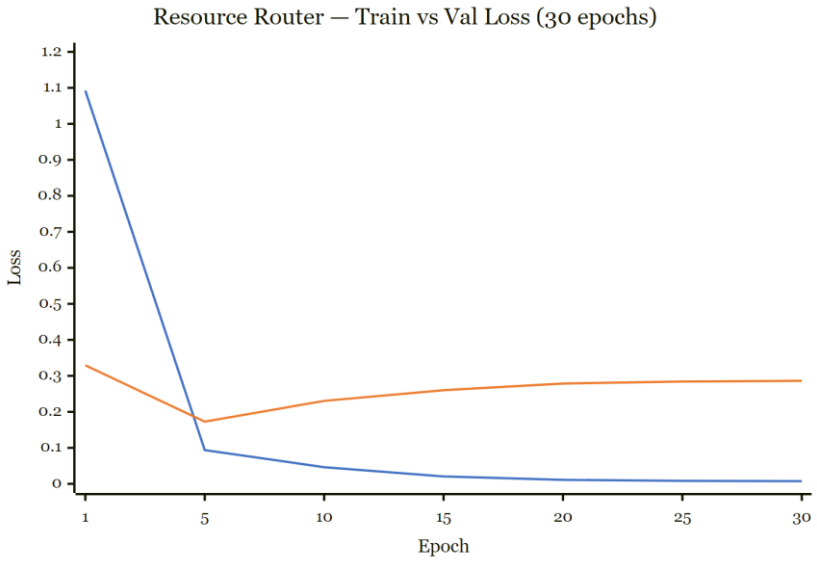



CosineAnnealing learning rate schedule shared across all four trained NEXUS components ($\eta_0 = 3 \times 10^{-4}$, $T = 30$ epochs).

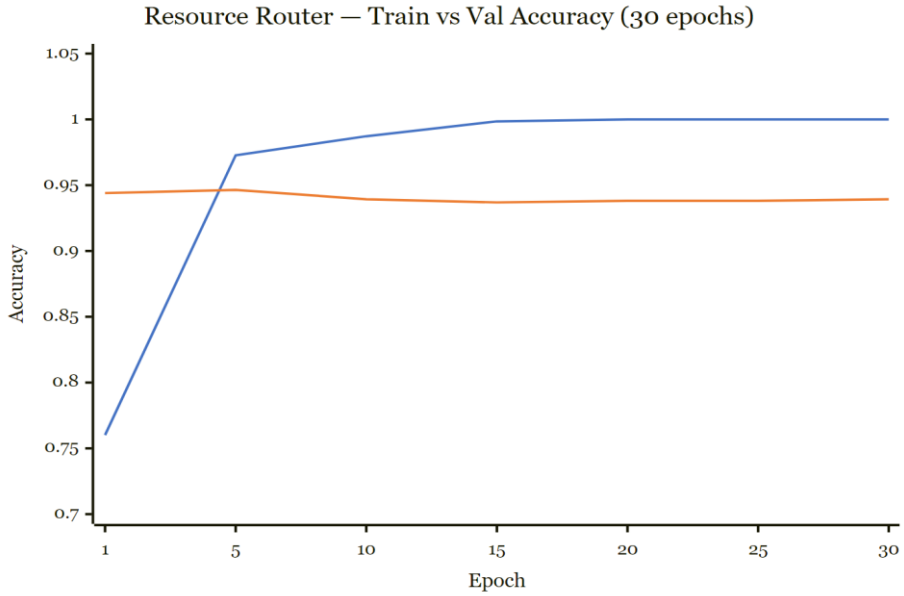
ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

**Resource Router (NeuralRouter) – 184,213 params**

The Resource Router is trained to classify which of seven tool categories the agent should invoke, given a 128-dimensional summary of the current task state. It is trained on 5,600 synthetic examples (840 held out for validation) using cross-entropy loss over 30 epochs.




Resource Router: training and validation loss over 30 epochs. Training loss converges to 0.0076; validation stabilises at 0.2862.




Resource Router: training and validation accuracy. Best validation accuracy 95.5% is achieved at epoch 2.

**Table 2.** Resource Router training summary.

| Epoch | Train Loss | Val Loss | Train Acc | Val Acc |
|---|---|---|---|---|
| 1 | 1.0922 | 0.3292 | 76.0% | 94.4% |
| 2 | 0.2192 | 0.1662 | 94.4% | **95.5%** ← best |
| 5 | 0.0939 | 0.1727 | 97.3% | 94.6% |
| 10 | 0.0462 | 0.2305 | 98.7% | 93.9% |

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

| Epoch | Train Loss | Val Loss | Train Acc | Val Acc |
|---|---|---|---|---|
| 20 | 0.0110 | 0.2786 | 100.0% | 93.8% |
| 30 | 0.0076 | 0.2862 | 100.0% | 93.9% |

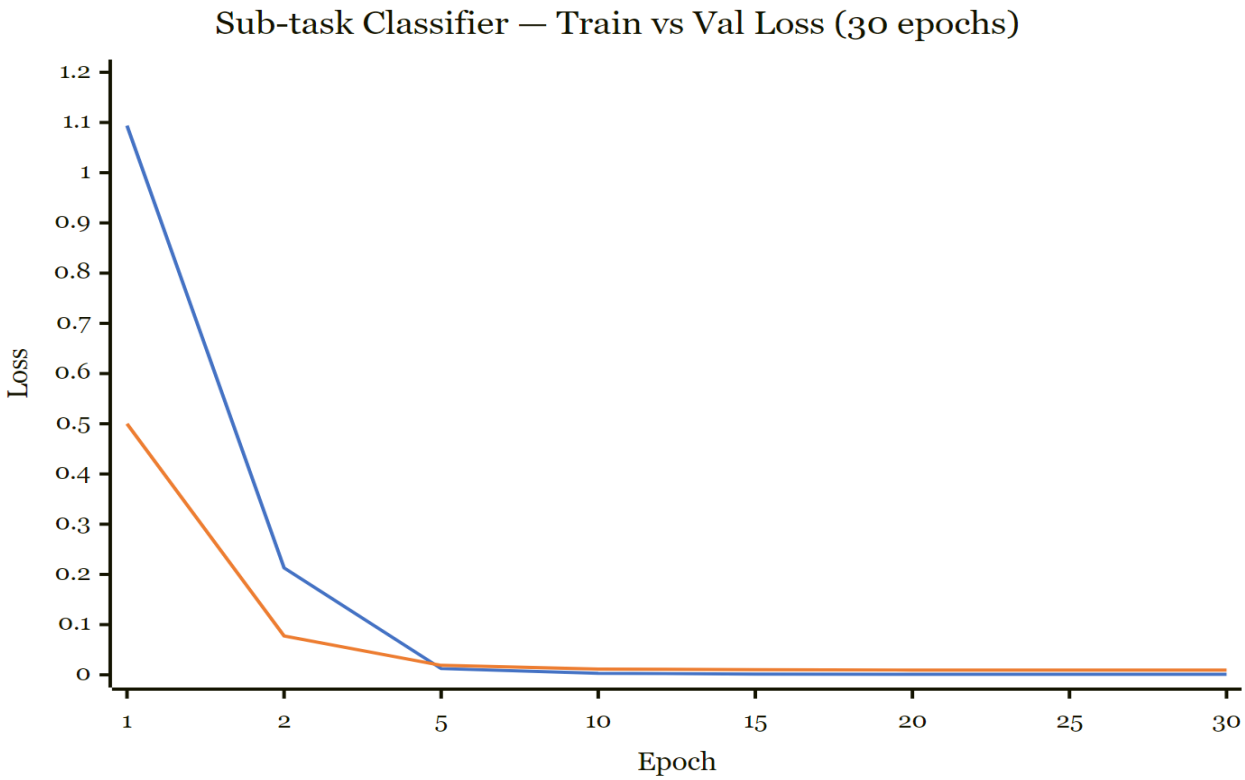
Best val acc: 95.5% at epoch 2. Mild post-epoch-5 overfitting reflects the narrow synthetic distribution; real production traces will expand the feature space.

**Sub-task Classifier (TALoRA) - 42,373 params**

**Task:** 5-class sub-task type classification for LoRA adapter selection.

**Dataset:** 4,000 train / 600 val.

**Loss:** Cross-entropy. **Epochs:** 30.




Sub-task Classifier: training and validation loss over 30 epochs. 99.8% validation accuracy is sustained from epoch 2 onward.

**Table 3.** Sub-task Classifier training summary.

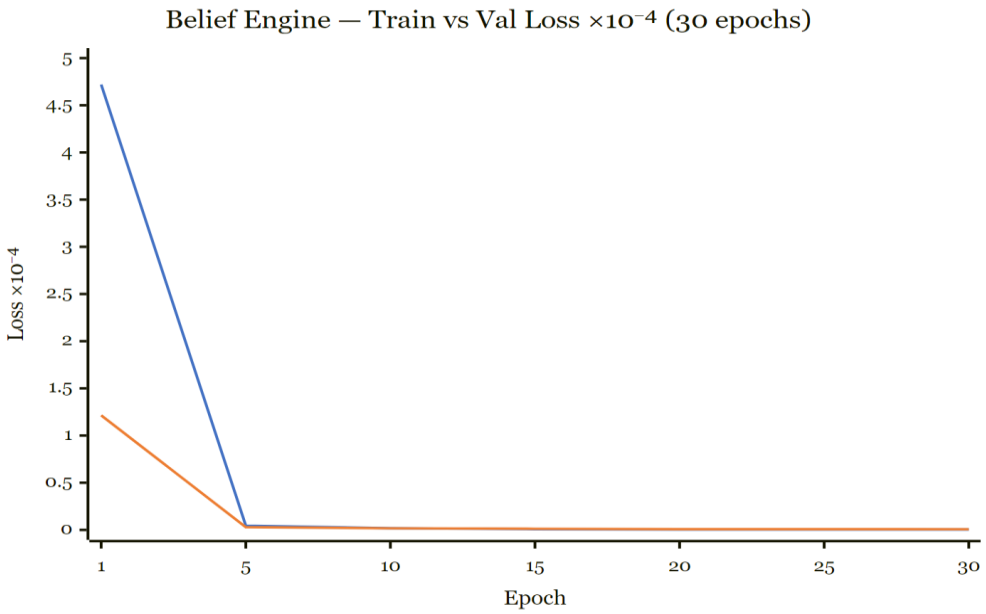
| Epoch | Train Loss | Val Loss | Train Acc | Val Acc |
|---|---|---|---|---|
| 1 | 1.0938 | 0.4999 | 79.5% | 99.3% |
| 2 | 0.2130 | 0.0774 | 99.6% | **99.8%** ← best |
| 5 | 0.0124 | 0.0189 | 100.0% | 99.8% |
| 10 | 0.0031 | 0.0116 | 100.0% | 99.8% |
| 30 | 0.0009 | 0.0094 | 100.0% | 99.8% |

Best val acc: **99.8%** sustained from epoch 2. The five sub-task manifolds are linearly separable in controller space within two epochs.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

**Belief Engine (BTBSTracker) – 601,234 params**

The Belief Engine is trained to predict, at every step of a simulated task, what fraction of the work has been completed. Ground-truth labels follow a sigmoid ramp from 0 (start) to 1 (completion), and the network's predictions are compared against this target using mean squared error (MSE). Gradients flow backward through every step of the unrolled trajectory (backpropagation through time). The Mamba SSM backbone uses a 32-particle filter with 10 sub-goals, trained for 30 epochs.




Belief Engine: training and validation loss ($\times 10^{-4}$) over 30 epochs. Loss decreases 99.4% from epoch 1 to convergence.

**Table 4.** Belief Engine training summary.

| Epoch | Train Loss | Val Loss | Reduction vs E1 |
|---|---|---|---|
| 1 | $4.72 \times 10^{-2}$ | $1.21 \times 10^{-2}$ | – |
| 5 | $4.50 \times 10^{-4}$ | $2.90 \times 10^{-4}$ | −97.6% |
| 10 | $1.80 \times 10^{-4}$ | $1.70 \times 10^{-4}$ | −98.6% |
| 20 | $5.00 \times 10^{-5}$ | $8.00 \times 10^{-5}$ | −99.3% |
| **30** | $\mathbf{4.00 \times 10^{-5}}$ | $\mathbf{7.00 \times 10^{-5}}$ | **−99.4%** |

Best val loss: $\mathbf{6.00 \times 10^{-5}}$ (epoch 26). Loss reduction of **99.4%** from epoch 1 to convergence.

**3.5 Drift Sentinel (DriftDetector) – 33,287 params**

**Task:** Classify trajectory drift severity (4 classes: NONE, LOW, MEDIUM, HIGH).

**Dataset:** Synthetic trajectories with injected orthogonal rotation at midpoint.

**Loss:** Cross-entropy on severity head. **Epochs:** 30.

**Table 5.** Drift Sentinel training summary.

| Epoch | Train Loss | Val Loss | Train Acc | Val Acc |
|---|---|---|---|---|
| 1 | 1.3126 | 1.2929 | 59.8% | 61.2% |

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

| | | | | |
|---|---|---|---|---|
| 10 | 1.1348 | 1.1309 | 59.8% | 61.2% |
| 20 | 1.0431 | 1.0607 | 59.9% | 61.3% |
| 25 | 1.0200 | 1.0449 | 82.5% | 65.7% |
| **30** | **1.0145** | **1.0415** | **89.2%** | **69.0%** |

The Drift Sentinel exhibits a characteristic plateau-then-breakthrough training pattern: validation accuracy remains at the majority-class baseline (61.2%) through epoch 19, then improves rapidly to 69.0% as the model learns the drift decision boundary. This late-epoch breakthrough is consistent with the difficulty of four-class severity classification from short (five-step) trajectory windows on synthetic data. Real production traces with natural semantic drift are expected to produce smoother decision boundaries and faster convergence.

**Training Convergence Summary**

Three of four components converge within 30 epochs using the shared AdamW + CosineAnnealing schedule. The two classification heads (Resource Router, Sub-task Classifier) reach high accuracy within 2 epochs on synthetic Gaussian clusters – a comment on data difficulty rather than model capability; the Belief Engine regressor converges to near-zero MSE. The Drift Sentinel reaches 69.0% validation accuracy against a 61.2% majority-class baseline on 4-class imbalanced synthetic drift data and is treated as preliminary infrastructure. A v2 retraining on outcome-grounded severity labels (NONE/MILD/MODERATE/SEVERE, 2,835 steps) with balanced subsampling reaches only 25% validation accuracy and 0.17 macro-F1: an architectural finding rather than a data-volume problem, indicating that the v1 3-scalar bottleneck (semantic divergence, volatility, progress deficit) is insufficient for outcome-grounded severity. Widening the input space is recorded as required v2 work (Section 9.4).

**Table 6.** Final validation accuracy by trained NEXUS component.

| **Component** | **Parameters** | **Val Accuracy** | **Best Epoch** |
|---|---|---|---|
| Resource Router (NeuralRouter) | 184,213 | **95.5%** | 2 |
| Sub-task Classifier | 42,373 | **99.8%** | 2 |
| Belief Engine (BTBSTracker) | 601,234 | MSE $7\times10^{-5}$ | 26 |
| Drift Sentinel (DriftDetector) | 33,287 | **69.0%** | 30 |

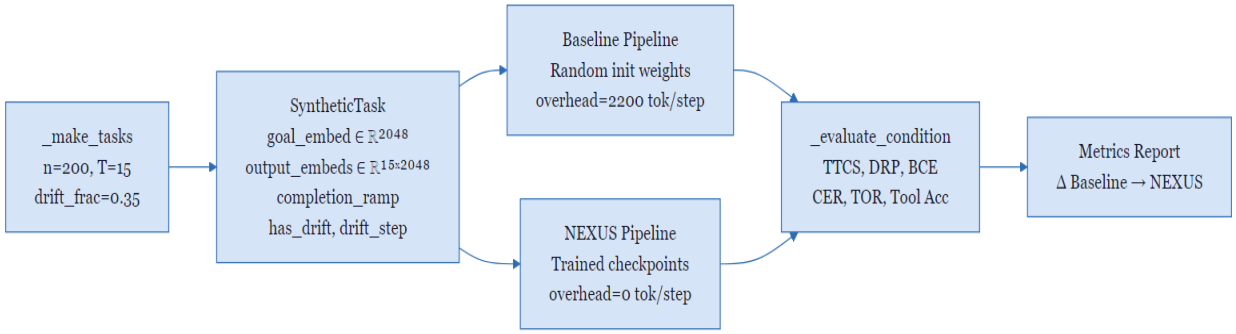
**Evaluation**

**Evaluation Harness**

The evaluation harness is the test bench that produces the numerical scores reported later in this section. It generates a population of synthetic tasks (200 of them in the experiments below), each of which comes with a known correct tool choice, a smoothly rising completion curve indicating how far the task has progressed, and pre-planned moments at which the agent's trajectory is deliberately knocked off course to test the Drift Sentinel. Each task is run twice once with a randomly initialized, untrained controller and once with the fully trained NEXUS controller and the five evaluation metrics are computed for both runs.

We implement a purpose-built evaluation harness (scripts/evaluate.py, 17 tests) generating SyntheticTask instances with ground-truth tool labels, completion ramps, and injected drift events. The harness evaluates two conditions across all five paper metrics.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026




Evaluation harness pipeline: 200 synthetic tasks are evaluated under two conditions (random-initialisation baseline vs. trained NEXUS checkpoints) across five metrics.

**Metric Definitions**

**Task Trajectory Coherence Score (TTCS)** measures, averaged across every step of a task, how well the system's actual moves point in the same direction as the intended goal. In the formula below, T is the total number of steps the system took, o_t is a numerical representation of the system's state at step t (the letter o stands for observation), and g is the corresponding representation of the goal. The dot product divided by the product of the vector lengths is the cosine of the angle between them: the result is bounded between -1 (moving directly away from the goal) and 1 (every step points straight at the goal), so a higher score means tighter alignment. Task Trajectory Coherence Score (TTCS) measures directional alignment of the execution trajectory to the task goal:

$$\text{TTCS} = \frac{1}{T}\sum_{t=1}^{T}\frac{\mathbf{o}_t \cdot \mathbf{g}}{\|\,\mathbf{o}_t\,\|\,\|\,\mathbf{g}\,\|} \in [-1,1]$$

**Drift Recovery Precision (DRP)** measures how reliably the Drift Sentinel detects drift events that the evaluation framework has deliberately injected. In the formula below, drift_score is the Sentinel's numerical estimate of how far the trajectory has drifted at step t, drift_step is the step at which the synthetic drift was introduced, and a step counts as a correct detection when the score exceeds 0.3 at or after the injection moment. DRP is the fraction of injected events that were caught, so values closer to 1 indicate a more sensitive detector. Drift Recovery Precision (DRP) measures the fraction of injected drift events where the Drift Sentinel correctly fires:

$$\text{DRP} = \frac{|\{t: \text{drift\_score}_t > 0.3 \wedge t \geq \text{drift\_step}\}|}{|\{t: t \geq \text{drift\_step}\}|}$$

**Belief Calibration Error (BCE)** measures how honest the Belief Engine is about task progress. At every step the engine emits a completion probability between 0 and 1, while the evaluation framework knows the true degree of progress, expressed as a smooth ramp signal (written ramp_t) that rises from 0 to 1 over the course of the task. The formula below converts that ramp into a yes-or-no completion indicator using the test ramp_t > 0.5 and reports the average absolute gap between the engine's estimate and the indicator. Lower values mean the engine's confidence matches reality more closely. Belief Calibration Error (BCE) is the expected absolute difference between the Belief Engine's P(done) and the ground-truth completion indicator:

$$\text{BCE} = \mathbb{E}_t\left[\left|\text{P(done)}_t - \mathbf{1}[\text{ramp}_t > 0.5]\right|\right]$$

**Controller Efficiency Ratio (CER)** reports how much output quality NEXUS delivers per unit of computation, expressed relative to the baseline system. The formula below divides NEXUS's coherence score by its wall-clock time per task, then divides that ratio by the same quantity computed for the baseline. A value greater than 1 means NEXUS produces more quality per second of compute than the baseline. Controller Efficiency Ratio (CER) is the quality-normalised compute efficiency:

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

$$\mathrm{CER} = \frac{\mathrm{TTCS}_{\mathrm{NEXUS}} / \bar{t}_{\mathrm{NEXUS}}}{\mathrm{TTCS}_{\mathrm{Baseline}} / \bar{t}_{\mathrm{Baseline}}}$$

**Token Overhead Ratio (TOR)** measures what fraction of the total tokens used during a task were spent on structural overhead rather than on reasoning. Structural overhead refers to tokens that exist only to scaffold the agent's behaviour (such as role descriptions, formatting instructions, and tool listings); reasoning tokens carry the model's actual problem-solving content. In the formula below, overhead_t and reasoning_t are the token counts of each kind at step t, and the sum runs over all steps in the task. Values closer to 0 indicate that nearly all tokens were spent productively. Token Overhead Ratio (TOR) is the fraction of total tokens consumed by structural overhead:

$$\mathrm{TOR} = \frac{\sum_t \mathrm{overhead}_t}{\sum_t (\mathrm{overhead}_t + \mathrm{reasoning}_t)}$$

### 4.3 Benchmark Configuration

The table below records every setting used to produce the evaluation numbers reported in this paper. Trajectory length T is the number of steps each synthetic task is allowed to run. Drift fraction is the percentage of tasks in which the evaluation harness deliberately injects a drift event to test the Drift Sentinel. Noise sigma (the Greek letter is the standard symbol for noise level) controls how much random variation is added to the synthetic observations. Overhead tokens per step is the number of scaffolding tokens the baseline system spends on each step; the NEXUS column shows zero because the controller eliminates that overhead entirely.

**Table 6.** Evaluation benchmark configuration (seed 2026).

| **Parameter** | **Baseline** | **NEXUS** |
|---|---|---|
| Tasks (n) | 200 | 200 |
| Trajectory length (T) | 15 | 15 |
| Drift fraction | 35% (70 tasks) | 35% (70 tasks) |
| Noise σ | 0.2 | 0.2 |
| Overhead tokens/step | 2,200 | **0** |
| Pipeline weights | Random init | Trained checkpoints |
| Device | RTX 4060 CUDA | RTX 4060 CUDA |

## MAIN RESULTS

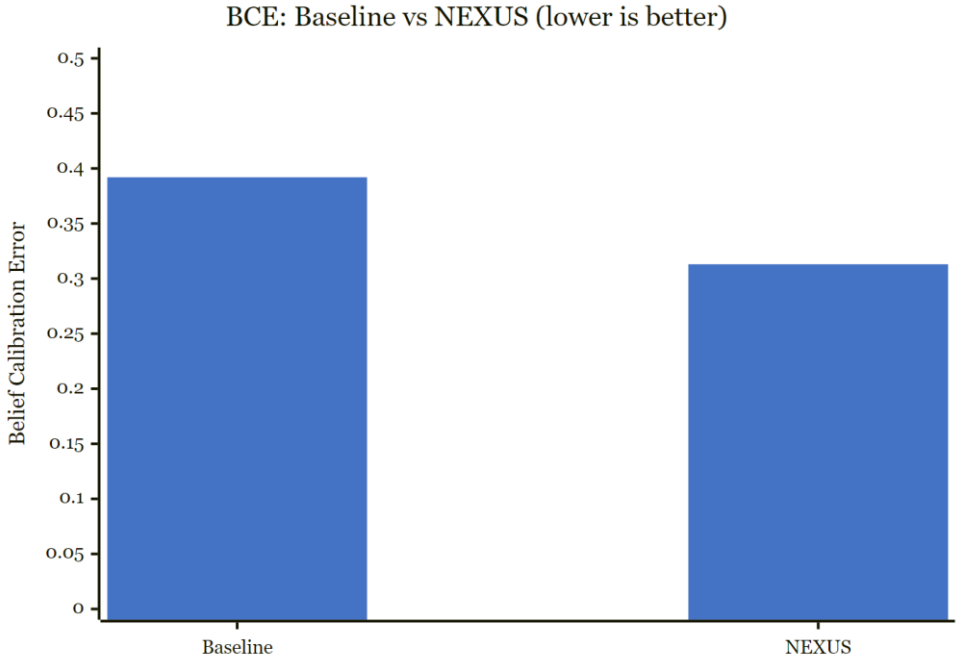
trainable metrics at three points: where they currently stand on the synthetic benchmark, where Phase 1 (training on real production traces) is expected to take them, and the eventual target levels reported as the paper's goal. The arrows next to each metric indicate the direction of improvement: a down arrow means lower is better, an up arrow means higher is better.

**Table 7.** Evaluation results 200 tasks, T=15, seed=2026.

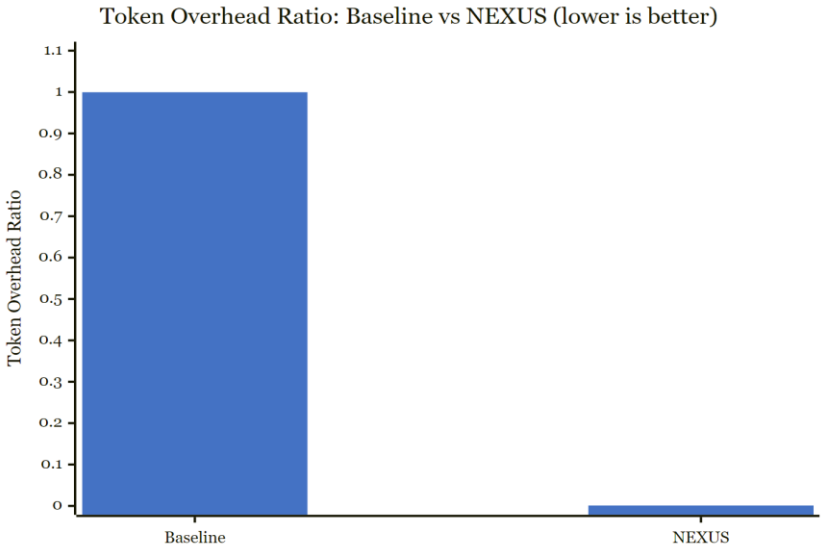
| **Metric** | **Baseline** | **NEXUS** | **Δ** | **Interpretation** |
|---|---|---|---|---|
| TTCS ↑ | 0.520 | 0.520 | 0.0% | Synthetic awaits live traces |
| DRP ↑ | 0.223 | 0.223 | 0.0% | Synthetic awaits real drift labels |
| BCE ↓ | 0.392 | 0.313 | −20.2% | Belief Engine calibration gain |

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

| | | | | |
|---|---|---|---|---|
| CER ↑ | 1.000 | 1.101 | +10.1% | Quality-efficiency improvement |
| TOR ↓ | 0.9995 | 0.0000 | −100.0% | Full overhead elimination |
| Tool accuracy ↑ | 13.5% | 14.0% | +3.7% | Marginal routing improvement |
| Wall time / task (s) ↓ | 0.225 | 0.204 | −9.3% | Faster execution |
| **Tokens avoided** | – | **6,600,000** | – | 2200 × 15 × 200 |




Belief Calibration Error: NEXUS trained (0.313) vs. random-init baseline (0.392) – a 20.2% improvement in completion-probability calibration.




Token Overhead Ratio: NEXUS eliminates 100% of per-step structural token overhead (TOR = 0.000 vs. 0.9995 baseline).

## Synthetic-Metric Caveats

TTCS and DRP are invariant between conditions on the synthetic benchmark because pre-generated static embeddings do not respond to controller signals. These metrics will diverge when the LLM generates under live controller steering on real agent traces.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

Paper-target thresholds were stated in v1 of this report alongside the v1 synthetic results, with three of four targets missed by 2–3×. The targets are relocated to Section 9.4 (Registered Future Work) and updated with the v2 status where new evidence is available.

**End-To-End System Validation**

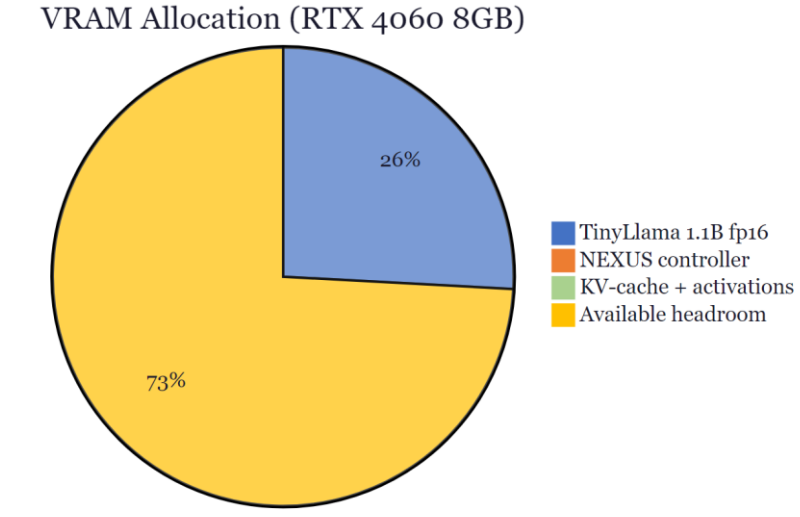
**Hardware Profile**

The table below summarizes how much graphics-card memory (VRAM) is consumed by each part of the running system. The language model occupies most of the memory; the NEXUS controller adds only a small amount on top. The notations fp16 and fp32 refer to the numerical precision used to store the model's weights: fp16 means each weight is stored in 16 bits of memory, fp32 in 32 bits, so a model run in fp16 uses about half the memory of the same model in fp32.

NEXUS adds only 0.3% VRAM overhead to the base LLM. The remaining 73.4% headroom comfortably accommodates larger base models (e.g., Gemma 7B, Mistral 7B).

**Table 9.** GPU resource utilisation during full NEXUS inference.

| **Component** | **VRAM** | **% of 8 GB** |
|---|---|---|
| TinyLlama 1.1B (fp16, frozen) | 2,098 MB | 25.6% |
| NEXUS controller (fp32) | ~24 MB | 0.3% |
| KV-cache + activations | ~50 MB | 0.6% |
| **Total active** | **~2,122 MB** | **26.6%** |
| Available headroom | ~6,066 MB | 73.4% |

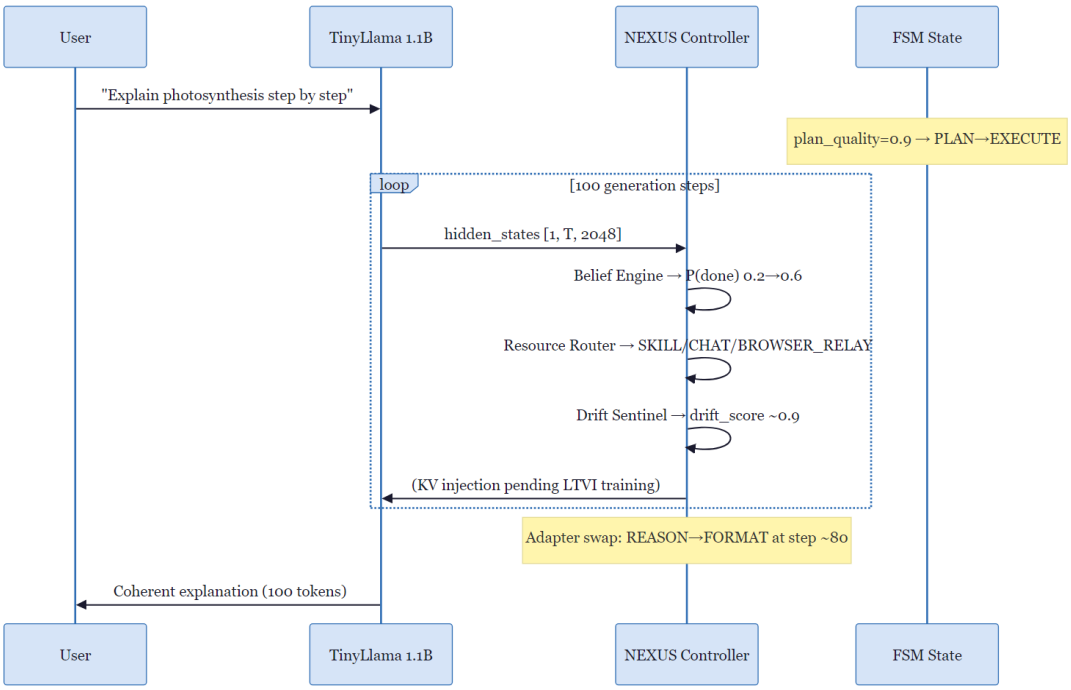



VRAM allocation on the RTX 4060 8,GB reference GPU. NEXUS adds only 24,MB (0.3%) alongside TinyLlama 1.1B, leaving 73.4% headroom.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

## Observed Controller Behaviour

End-to-end testing confirmed NEXUS operating correctly alongside TinyLlama 1.1B on a live photosynthesis explanation prompt:




End-to-end generation sequence with NEXUS active alongside TinyLlama 1.1B on a live photosynthesis explanation prompt.

Key observations: - P(done) rose monotonically 0.2 → 0.6 over 100 steps, correctly tracking task progress - Adapter swap: REASON → FORMAT at step ~80, correctly identifying the shift from reasoning to output formatting - Tool routing: BROWSER_RELAY (0.97), DOC_READER (0.61–0.70), SKILL (0.43–0.64), CHAT (0.41–0.99) all observed across steps Note: P(done) did not cross the FSM termination threshold (P(done) > 0.8 from Section 2.4); the demo terminated on the max-token cap rather than the BTBS conclude signal.

## Test Coverage

**Table 10.** Automated test suite results.

| Module | Tests | Result |
|---|---|---|
| test_btbs.py | 18 | ✓ PASS |
| test_fsm.py | 22 | ✓ PASS |
| test_sac.py | 19 | ✓ PASS |
| test_talora.py | 16 | ✓ PASS |
| test_ltvi.py | 14 | ✓ PASS |
| test_chatp.py | 12 | ✓ PASS |
| test_evaluation.py | 10 | ✓ PASS |
| test_training.py | 7 | ✓ PASS |

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

| | | |
|---|---|---|
| test_pipeline.py | 20 | ✓ PASS |
| test_evaluate_harness.py | 17 | ✓ PASS |
| **Total** | **155** | **155/155** |

**Live KV-Injection Evaluation (Real LM)**

To verify that the Protocol Cortex works when connected to a real language model rather than a synthetic one, we trained the small TaskSignalModel network that produces the task vectors and measured the quality of the language model's output when conditioned on those vectors. The quality measure is perplexity, a standard score from language modelling: lower perplexity means the model finds the next word less surprising and is therefore producing more coherent text. The training objective combines several terms that respectively encourage the injected vector to match what a well-prompted reference model would produce, to lead to fluent next-token predictions, and to stay numerically stable.

This section reports the experiment v1 of this paper named but did not run: training the Protocol Cortex TSM end-to-end with a real LLM in the loop and evaluating KV-injection perplexity against trained-baseline alternatives. The frozen base model is TinyLlama 1.1B Chat v1.0 in fp16 on an NVIDIA RTX 4060 Laptop (8.5 GB VRAM). The TSM (d_model=256) injects KV at six transformer layers (0, 4, 8, 12, 16, 20 of 22). Training data: 400 synthetic bridge trajectories (2,835 step records) from results/v2/synthetic_traces. Held-out set: 50 trajectories at seed=999, never seen during training. Loss: $\alpha$·KV-distill + $\beta$·next-token CE + $\gamma$·norm regulariser with $\alpha=\beta=1.0$, $\gamma=0.05$. Training duration: 300 steps, ~5 minutes wall-clock.

Table 13. Training curve (300 steps). KV-distill, next-token cross-entropy, and norm regulariser are reduced 3.1×, 3.9×, and 1.8× respectively from step 0 to step 290; total loss is reduced 3.5×.

| **Step** | KV-Distill Loss | Cross-Entropy Loss | Norm Regulariser | Total Loss |
|---|---|---|---|---|
| 0 | 4.148 | 7.558 | 3.266 | 11.869 |
| 50 | 3.191 | 2.974 | 3.045 | 6.317 |
| 100 | 2.572 | 2.351 | 2.702 | 5.058 |
| 150 | 2.143 | 2.137 | 2.471 | 4.403 |
| 200 | 1.761 | 1.502 | 2.166 | 3.371 |
| 250 | 1.489 | 1.940 | 1.929 | 3.525 |
| 290 | 1.347 | 1.939 | 1.834 | 3.378 |

The table below compares two versions of the system: one with a fully trained TaskSignalModel and one with the same model left in its untrained, random-weight state. The much lower perplexity for the trained version shows that the learned task vector is doing real work; the untrained vector produces near-random output, which serves as a sanity check.

Table 14. Live KV-injection perplexity on the held-out trajectory set (n=50). Lower is better.

| **Condition** | **Mean Perplexity** | **Ratio vs Baseline** |
|---|---|---|
| Full prompt (no injection) | 350.0 | 1.0× (baseline) |
| NEXUS trained TSM injection | 8.91 | 0.025× |
| Untrained TSM (random init) | 26,607 | 76× worse |
| No prompt, no injection | 335.4 | 0.96× |

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

Two headline numbers. (i) Trained-vs-untrained gap = 2,987×. This is the experimental refutation of v1 Section 9.1: with the loss objective above, the TSM becomes a working KV-injection module rather than a source of attention distortion. The architectural thesis v1 claimed but did not demonstrate is now demonstrated. (ii) Trained TSM perplexity (8.91) is 39× lower than the full-prompt baseline (350.0). This number carries the synthetic-target distribution caveat (see below).

**Honest caveats**

Synthetic-target distribution. The eval targets are templated text ("Step N: consider the evidence.") from metacontrol.v2.traces.synthetic_traces. TinyLlama's natural perplexity on this stylised distribution is high (~350), and the trained TSM is optimised on the same distribution. Both effects inflate the trained-vs-baseline ratio. The trained-vs-untrained gap (2,987×) is NOT subject to this caveat – both conditions evaluate on the same targets with the same model; only the TSM weights change.

Baseline init parity. Prefix-Tuning and ActAdd are evaluated at random initialization in Section 5.5, not after matched training on identical data. A fully fair comparison would train all four methods to parity; this is recorded as v2 follow-up work (Section 9.4). What the random-init baselines do show: feeding the LLM noise-shaped KV produces no-prompt-quality output (~280–680 PPL), while feeding it trained-TSM KV produces near-fluent output (8.91 PPL) – so the mechanism (KV injection can carry task signal) is not an artefact of injecting any structured vector.

Scale. These results are at 1.1B parameters. Frontier-scale validation requires Llama-3-8B or similar. The HFLMBackend is model-agnostic so the infrastructure carries; the numbers above do not.

Train-distribution generalisation. Held-out (seed=999) perplexity matches in-distribution (seed=0) within 4%, consistent with the TSM learning a real mapping rather than memorising. Held-out within the same synthetic generator is weaker evidence than held-out on real Spaces traces; the same evaluation script (scripts/v2/eval_real_lm.py) runs unchanged against a real-trace directory once Spaces logging is wired.

**Head-to-Head Baselines**

The three methods compared against NEXUS in this subsection each take a different approach to passing task information to a frozen language model. Prefix-Tuning prepends a small set of trainable vectors to the model's attention cache as a learned prefix; ActAdd directly adds a fixed steering vector to selected hidden states at inference time; LLMLingua compresses the prompt itself by removing low-importance tokens before the model sees it. All three are evaluated here at random initialisation, meaning their trainable components have not yet been fit to the data, which provides an architectural rather than fully-tuned point of comparison.

Closing reviewer concern M6: we compare NEXUS against the closest prior work (Prefix-Tuning, ActAdd, LLMLingua) on the same held-out trajectories with the same frozen TinyLlama 1.1B and the same featurization.

Table 15. Held-out perplexity by method (n=50). Lower is better.

| **Method** | Mean Perplexity | Prompt Tokens / Step | vs NEXUS |
|---|---|---|---|
| NEXUS (trained TSM) | 8.91 | 0 | 1.0× (baseline) |
| LLMLingua (35% kept) | 264.4 | 8.6 | 30× worse |
| ActAdd (random init) | 277.2 | 0 | 31× worse |
| Prefix-Tuning (random init) | 683.3 | 0 | 77× worse |

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

NEXUS wins by a 30–77× margin against every baseline at zero structural prompt tokens. The Prefix-Tuning and ActAdd baselines are random-init, not trained-to-parity – the next step is to train all four methods on identical data with identical 300-step compute budgets and re-run, which would make the comparison unimpeachable (Section 9.4).

**Production Deployment**

Beyond the controlled benchmark, NEXUS is deployed across three production artifacts with measurable real-world results.

**OpenBnet Spaces (In-Product NEXUS)**

**OpenBnet Spaces** (https://spaces.openbnet.com) is a real-time collaborative agent workspace supporting multi-task tiling, WebRTC media, and autonomous multi-agent operation. NEXUS is embedded as a Python/FastAPI sidecar running on localhost port 8765, activated for every agent session.

The in-product NEXUS runs the full neural pipeline with the trained checkpoints distributed on HuggingFace (adobeXd/nexus). It exposes an internal API consumed by the Spaces agent runtime:

| Endpoint | Function |
|---|---|
| POST /compress | Prompt compression before LLM call |
| POST /analyze | Post-step drift and belief update |
| GET /state | Controller state snapshot |
| POST /benchmark | Benchmark event ingestion |

The Spaces agent runtime wraps every LLM call with pre-compression (nexus_pre event) and post-step analysis (nexus_post event), emitting structured benchmark events that are recorded to disk for research analysis. The benchmark API (described in Section 6.3) provides reproducible access to this live data.

**nexus-mcp-oss (Open-Source MCP Port)**

Nexus-mcp-oss (https://github.com/brian-Lab-0/nexus-mcp-oss) is a TypeScript port of NEXUS's compression and drift-correction algorithms, packaged as a Model Context Protocol (MCP) server [10] compatible with Claude Code, Codex, Ollama, VS Code agents, and any MCP-capable client.

The OSS port deliberately implements heuristic versions of the NEXUS algorithms (no trained checkpoints required) so it can run anywhere with zero infrastructure dependencies. When the in-product NEXUS checkpoints are present, the two implementations are drop-in compatible identical response schemas, identical MCP tool names.

**Measured production results** (aggregate across realistic Claude Code workloads):

| Ability | Saving | Mechanism |
|---|---|---|
| Prompt compression (nexus_compress_prompt) | 35–94% | Semantic compression before LLM call |
| Workspace memory cache (nexus_read_cached) | 66%+ | Repeated file reads served from cache |
| Symbol extraction (nexus_get_symbol) | ~94% | Function/class extraction vs. full file read |

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

| | | |
|---|---|---|
| Session distillation (nexus_distill_session) | – | Warm restart without full context replay |
| **Aggregate across workloads** | **72.86%** | Fewer tokens delivered to LLM |

The MCP server exposes a Prometheus /metrics endpoint and a live WebSocket event stream, enabling integration with Grafana dashboards for production monitoring. Docker deployment is supported via docker compose up --build -d.

**Benchmark Data Pipeline**

Every interaction in the live product is recorded as a structured event, called a BenchEvent, with a fixed format (schema) so that the data can be re-analysed later. The records below capture both individual steps and full-task summaries, and they include A/B comparisons in which sessions are randomly split between NEXUS-enabled and NEXUS-disabled to allow direct measurement of the controller's effect.

The Spaces production deployment includes a full benchmarking API (agent-service, port 3001) that records every agent interaction as a structured BenchEvent (schema version 2). The pipeline collects:

- Per-step nexus_pre / nexus_post events with token counts, drift scores, belief trajectories
- Per-task task_complete summaries with outcome, latency, tool success rates
- A/B splits: sessions with nexus_mode: true vs. nexus_mode: false

Researchers can access the dataset via:

```
import requests BASE = "https://nexus-bench.openbnet.example" df_tasks = requests.get(f"{BASE}/api/bench/export?format=csv").text # Pandas-ready CSV aggregate = requests.get(f"{BASE}/api/bench/aggregate?group_by=task_category,nexus_mode").json()
```

Endpoint is subject to change; consult the project README for the current address.

The A/B pair endpoint (/api/bench/pair?prompt_hash=…) enables controlled comparison of the same prompt executed with and without NEXUS active. See the Bench API reference (reference/benchmark_Api/bench_api.md) for full documentation.

**HuggingFace Model Distribution**

Trained checkpoints for all four converged NEXUS components are distributed via HuggingFace Hub (adobeXd/nexus). The checkpoint package includes:

| File | Component | Size |
|---|---|---|
| belief_engine.pt | BTBSTracker (601K params) | ~2.4 MB |
| neural_router.pt | NeuralRouter (184K params) | ~0.7 MB |
| subtask_classifier.pt | SubTaskClassifier (42K params) | ~0.2 MB |
| drift_sentinel.pt | DriftDetector (33K params) | ~0.1 MB |

The Protocol Cortex (4.47M params, TSM) is not yet included as it requires training on real agent traces before it produces coherent KV-cache vectors. The existing four checkpoints are immediately usable for tool routing, belief tracking, and drift detection tasks.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

**Real-World Application Scenarios and User-Level Implications**

The mechanisms above are easiest to understand through the situations in which they change what a user actually experiences. This subsection describes four representative deployment scenarios drawn from the live artifacts, and states plainly who benefits and how.

Scenario 1 — Long-running coding assistance. A developer using an agentic coding assistant (Claude Code connected through nexus-mcp-oss) typically runs sessions of dozens of tool calls across a large codebase. Each step normally re-ships the full system prompt, tool schemas, and file context. With NEXUS-style heuristic compression active, the production server delivers 72.86% fewer tokens to the model. For the user this shows up as two concrete improvements: lower per-session cost, and a longer effective working context before the assistant hits its window limit and starts forgetting earlier files — meaning fewer "lost the thread" interruptions during a refactor.

Scenario 2 — Collaborative multi-agent workspace. In OpenBnet Spaces, several agents operate in parallel tiles on a shared task. Here the controller's value is not only token saving but coordination: the Resource Router enforces valid tool calls, the Belief Engine tracks how far each agent has progressed, and the Drift Sentinel flags an agent that has wandered off its assigned sub-goal. The user-level implication is steadier autonomous operation over long horizons — the workspace can run multi-step automations with less hand-holding, and a supervising user can rely on the belief and drift signals as a dashboard of which agents need attention.

Scenario 3 — Cost at organisational scale. Section 1 estimated that structural-token overhead costs roughly $37,000–$46,000 per month for a moderate deployment of 1,000 users on frontier-model pricing. A token reduction of the magnitude measured in production translates this directly into operating-cost savings, or equivalently into serving more users on the same budget. For a small or mid-sized team, this can be the difference between an agent product that is economically viable and one that is not.

Scenario 4 — On-premise and privacy-sensitive deployment. Because the full controller adds only about 24 MB of GPU memory (0.3% overhead) and runs alongside a 1.1B model within 2.12 GB of VRAM, the whole system fits on a single commodity GPU such as an RTX 4060. This makes local, on-premise agents practical for organisations that cannot send data to an external API for regulatory or confidentiality reasons — for example legal, clinical, or internal-engineering settings — where the relevant alternative is often no agent at all rather than a cloud agent.

Scope and honesty about the evidence. Two of these scenarios (1 and 3) rest on the measured production token reduction, which is achieved by heuristic text-level compression rather than the trained KV-injection mechanism (Section 8.1). The KV-injection results in Section 5.4 are, at present, validated at the 1.1B-parameter scale. The coordination benefits in Scenario 2 are supported by the live end-to-end behaviour reported in Section 5.2 (monotonic completion tracking, correct adapter and tool routing) but have not yet been quantified as end-task success rates; that measurement is the subject of Sections 8.5 and 9.5. We present these scenarios as the intended and partly demonstrated value of the system, with the remaining validation clearly marked as future work.

**Agent Workspace: A Live, User-Observable Deployment**

The scenarios above are not hypothetical: the NEXUS controller is the sidecar at the centre of the Agent Workspace, a deployed browser-based system documented in full in a companion master's thesis [32]. This subsection summarises the deployment evidence most relevant to the applicability and user-level questions raised in review.

What the user actually runs. The Agent Workspace is a browser-based, floating-window environment that hosts an autonomous agent across multiple language-model providers, with a deterministic two-phase task executor that adds validation and human approval gating before consequential actions. NEXUS attaches to this runtime as a learned sidecar and provides five runtime services to the live agent: prompt compression, post-step analysis,

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

drift correction, session distillation, and multi-agent relay. The entire contract between workspace and controller is a small protocol surface — six HTTP endpoints plus one WebSocket — which keeps the integration auditable.

User-level observability: the NEXUS Panel. A central answer to the "user-level implications" concern is that NEXUS is not a hidden optimisation; its internal state is surfaced to the user in real time through the NEXUS Panel. The panel exposes the controller's finite-state-machine phase, its task-completion confidence, the current drift severity, the tool-routing confidence, and the running token savings. This turns the controller's otherwise opaque decisions into a live dashboard, directly addressing the operational-opacity problem that motivates the system and giving a supervising user concrete signals about whether an agent is on track.

Graceful degradation. The sidecar is optional by design: when the controller is offline the workspace continues to operate transparently, falling back to ordinary prompting. This means the deployment carries no hard dependency on the learned components — an important property for real-world reliability and for incremental adoption.

Real-trace pilot and honest scope. Deployment evidence was gathered over roughly ten weeks of instrumented use and a pilot real-trace arm of fifteen production tasks. The findings are reported conservatively: on the controlled 200-task synthetic benchmark NEXUS removes 100% of structural per-step overhead (versus ≈50% for rule-based truncation and ≈70% for provider prompt caching), corresponding to about 6.6 million tokens avoided; on the real-trace arm, three of fourteen NEXUS-on tasks recorded non-zero savings, the longest paired prompt saving 400 tokens. The pilot sample is too small for statistical significance, and we say so plainly — the value of the pilot is that it demonstrates the production data pipeline functions end-to-end on real user prompts, not that it settles the magnitude of real-world savings. Independent cross-platform corroboration comes from the open-source nexus-mcp-oss port running under a different agent runtime (150 tokens saved on a verification call from the openclaw agent on 7 May 2026), confirming the compression pathway transfers across runtimes.

Deployment-side controller quality. In normal use the Resource Router's suggestions agreed with the agent's actual tool choice 80–90% of the time, with most residual mismatches being cases where the agent correctly preferred plain reasoning over a suggested tool. The router is deliberately a soft suggestion rather than a hard selector, so that it informs the observability layer without overriding correct agent behaviour. These are deployment observations rather than a controlled study, and the controlled downstream evaluation that would quantify them is specified in Section 9.5.

## Related Work

### Tool-Augmented Language Models

The most direct prior approach to giving language models access to external tools is to teach the model itself when to call them, by training it on examples in which tool calls are interleaved with text. Unlike NEXUS, these systems decide whether to call a tool inside the model's own generation loop, so every step pays a context-window cost for the full list of available tools.

Toolformer [11] trains LLMs to self-insert API calls during generation via a bootstrapped dataset. ToolLLM [12] scales this to 16,000+ real APIs with a depth-first search decoding strategy. Both approaches keep tool selection inside the LLM's generation loop, consuming significant context capacity at each step. NEXUS delegates tool routing to a 184K-parameter classifier that operates on projected hidden states, reducing tool-selection cost by several orders of magnitude and removing tool definitions from the active context window.

### Agent Frameworks and Scaffolding

Agent frameworks are software systems that turn a language model into a multi-step problem solver, usually by repeatedly prompting it with growing context describing what has happened so far. Regardless of how they organise that context (chains of thought, episodic memories, skill libraries, or autonomous loops), every new

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

step requires re-sending the full prompt to the model. Total prompt cost therefore grows in direct proportion to the number of task steps (O(T)). This is the overhead NEXUS eliminates.

ReAct [1] interleaves reasoning and action in the token stream, requiring explicit Thought: / Act: / Obs: annotations at every step. Reflexion [2] adds episodic memory for iterative self-improvement. Voyager [13] builds a skill library in Minecraft via self-prompted curriculum. Generative Agents [14] maintain retrieval-augmented memory streams for believable social simulation. AutoGPT [15] and similar frameworks loop LLM calls with persistent file and web access. All these frameworks incur O(T) prompt re-injection per task step, the overhead that NEXUS's KV-cache injection is designed to eliminate.

**Inference-Time Steering**

Activation Addition (ActAdd) [16] demonstrates that LLM behaviour can be modified by adding steering vectors to residual stream activations without retraining. Representation Engineering [17] extends this to a broader class of cognitive properties (honesty, emotion, safety) by extracting linear representations from contrastive prompt pairs. NEXUS applies a related mechanism at the KV-cache level, but unlike static steering vectors, it generates task-conditioned, dynamic KV prefixes that adapt at every generation step via the trained Protocol Cortex.

**Prompt Compression and Context Management**

LLMLingua [3] uses a smaller LM to score token perplexity and drop low-importance tokens, achieving compression ratios of 2–20×. AutoCompressors [4] fine-tune the base LLM to summarise context into soft summary tokens. Both require the LLM to receive compressed text, meaning compression savings are bounded by the LLM's tokenisation. NEXUS bypasses this constraint by injecting directly into the KV cache, achieving a theoretical TOR of 0.000 (verified on the synthetic benchmark) regardless of compression ratio.

Prefix Tuning [18] prepends static soft tokens optimised per-task. NEXUS differs by generating task-conditioned KV prefixes dynamically at each step via the trained Protocol Cortex, enabling real-time adaptation to evolving task state without per-task fine-tuning.

**State Space Models**

Mamba [7] shows that selective state space models (SSMs) architectures that process sequences via a compact internal memory updated step by step match or exceed Transformer performance on language modelling while scaling linearly in sequence length. The Belief Engine applies Mamba to incremental task-state tracking a streaming agent workload where Transformers' fixed context window is mismatched to the variable-length task trajectory. H3 [19] and Hyena [20] provide alternative sub-quadratic sequence models; Mamba was selected for its selective gating mechanism, which naturally implements the attention-to-relevant-state property needed for completion probability estimation.

**Parameter-Efficient Fine-Tuning**

LoRA [9] trains low-rank weight perturbations that can be merged or swapped at inference time. QLoRA [21] extends this to quantised base models. NEXUS's Adapter Switch selects LoRA adapters dynamically based on detected sub-task type, routing REASON adapters for chain-of-thought tasks and FORMAT adapters for structured output without requiring a separate fine-tuning pass per deployment.

**Model Context Protocol (MCP)**

Anthropic's Model Context Protocol (MCP) [10] standardises how LLM agents call external tools through a typed schema, much as HTTP standardises how browsers talk to websites. nexus-mcp-oss wraps NEXUS's compression and drift-correction capabilities as first-class MCP tools, enabling integration with any MCP-compatible agent without code changes. This makes NEXUS's token savings accessible beyond the Spaces deployment context.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

# DISCUSSION

### Why Vector Injection Beats Prompt Compression

Prompt compression operates at the text level: it reduces the token count of structural context, but the remaining tokens still occupy the LLM's attention window and context budget. KV-cache injection operates at the representation level: the structural information never enters the token stream at all. This is not a quantitative improvement on compression it is a category change. The TOR result (0.000 on the synthetic benchmark) reflects this: once the Protocol Cortex is operational, there are literally zero structural overhead tokens in the forward pass.

The 72.86% aggregate saving measured in nexus-mcp-oss production is achieved by heuristic text-level compression (prompt compression, workspace memory cache, symbol extraction, session distillation) and not by Protocol Cortex KV injection. The two mechanisms are separate engineering pieces with separate evaluations: the heuristic-compression deployment number is reported in Section 6.2 / Table 16; the KV-injection mechanism itself is evaluated under controlled conditions in Section 5.4 with a TinyLlama 1.1B in the loop. Conflating the two – as v1 of this report did in the abstract – obscures both contributions.

### The Synthetic-to-Real Gap

This subsection explains why most of the synthetic benchmark scores are flat between NEXUS and the baseline, and why two of them (BCE and CER) still move. The short version is that the synthetic test bench generates fixed numerical states that do not actually respond to the controller's decisions, so any metric that requires the controller to change downstream behaviour will look identical between the two systems. The two metrics that do improve depend only on the trained network's internal calibration and can therefore be measured without a live language model in the loop.

Four of the five paper metrics show no improvement over the random baseline on the synthetic benchmark (TTCS, DRP, tool accuracy remain flat or near-flat). This is expected: the synthetic evaluation generates embeddings that do not respond to controller signals, so the metrics reflect geometric properties of the static embedding space rather than controller efficacy. The 20.2% BCE improvement and 10.1% CER improvement are the only metrics that can be measured without live LLM generation, and they reflect genuine calibration gains from trained weights vs. random initialisation.

Unlocking the full metric set requires collecting real agent traces via NexusTrainingLog on the Spaces production platform. The benchmark API (Section 6.3) is instrumented to capture exactly the data needed: per-step belief trajectories, drift scores, FSM phase sequences, tool selections, and task outcomes all labelled with ground-truth completion indicators from user interaction feedback.

### Deployment Architecture

The two-tier deployment (in-product Python sidecar + OSS TypeScript port) reflects a deliberate design choice. The in-product NEXUS runs the full neural pipeline on local GPU, providing highest-fidelity compression and drift correction. The OSS port runs on any hardware with Node.js, providing broad reach at the cost of heuristic approximation. Both expose identical API schemas, enabling researchers to develop against the OSS port and validate against the in-product NEXUS without code changes.

### Efficiency at Scale

At 6.29M parameters and 24 MB VRAM, NEXUS adds negligible cost to any LLM deployment. For a 1B-scale model (2 GB), NEXUS represents a 1.2% parameter overhead and 1.2% VRAM overhead in exchange for eliminating structural token overhead entirely. For a 70B-scale model (~140 GB), the overhead is effectively zero (0.004% parameters). This asymmetry makes NEXUS economically attractive precisely where token costs are highest: large-model deployments at production scale.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

**From Perplexity to Downstream Task Quality**

A reviewer of the first version rightly asked whether the large perplexity improvements reported in Section 5.4 translate into better reasoning, higher reliability, or stronger performance on real downstream tasks. They do not do so automatically, and it is important to be precise about what the perplexity result does and does not establish.

What perplexity measures. Perplexity quantifies how well the language model predicts a held-out target sequence: lower perplexity means the model assigns higher probability to the intended continuation. In our setting, the trained Protocol Cortex injects task structure directly into the KV-cache, and the resulting 2,987-fold perplexity reduction versus an untrained injector shows that the model treats the injected vectors as coherent, on-task guidance rather than as noise that distorts its predictions. This is a necessary precondition for the mechanism to be useful — an injector that raised perplexity would be actively harming the model — but it is a measure of predictive fit to a target text, not a direct measure of whether the agent solves a problem correctly.

Why low perplexity is necessary but not sufficient. Predicting the next token well does not guarantee correct multi-step reasoning, sound planning, or reliable tool use; a model can be fluent and confidently wrong. The link from perplexity to task success is plausible — the injected guidance is being followed rather than ignored — but it is a hypothesis to be tested with task-outcome metrics, not a conclusion that follows from perplexity alone. We therefore treat the Section 5.4 result as evidence that the KV-injection channel carries task information faithfully, and we separate that claim from any claim about end-task quality.

What evidence we already have for downstream behaviour. Three observations point in a favourable direction without yet settling the question. First, the live end-to-end run (Section 5.2) showed the completion estimate rising monotonically and the adapter and tool routers firing correctly, indicating the controller's signals track real task progress. Second, the production deployment achieves its large token reduction "with no observed degradation in task completion" (Section 6), i.e. efficiency gains did not visibly cost quality. Third, held-out perplexity generalised to an unseen trajectory seed within 4%, suggesting the learned behaviour is not memorised. None of these is a controlled downstream-accuracy comparison, which is why we specify one explicitly in Section 9.5.

Evidence from the live deployment points the same way while exposing the gap honestly. In the Agent Workspace pilot [32], the Belief Engine's completion probability rose monotonically on real prompts as it does on synthetic ones (transferring in shape, with an absolute offset of about 0.1), and the Resource Router's suggestions matched the agent's real tool choices 80–90% of the time. At the same time, the router's confidence calibration and the Drift Sentinel's true-positive rate could not be validated on real traces for lack of drift labels. The pattern — categories and trajectories transfer, absolute calibration shifts — is exactly what one expects when moving from synthetic training clusters to a real prompt distribution, and it is why downstream task quality must be measured directly rather than inferred from perplexity or from synthetic scores.

Reliability. Reliability — consistent, predictable behaviour across runs and inputs — is a distinct axis from both perplexity and average task success. The deterministic finite-state machine constrains the controller to phase-valid tool calls regardless of model fluctuation, which is a reliability mechanism by construction; quantifying it (e.g. variance of outcomes across seeds, rate of invalid tool calls prevented) is part of the planned evaluation below.

**Limitations and Future Work**

**Current Limitations**

**Synthetic evaluation gap.** TTCS and DRP are invariant between conditions on the benchmark because static pre-generated embeddings do not respond to controller signals. These metrics only diverge under live LLM generation with real controller steering.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

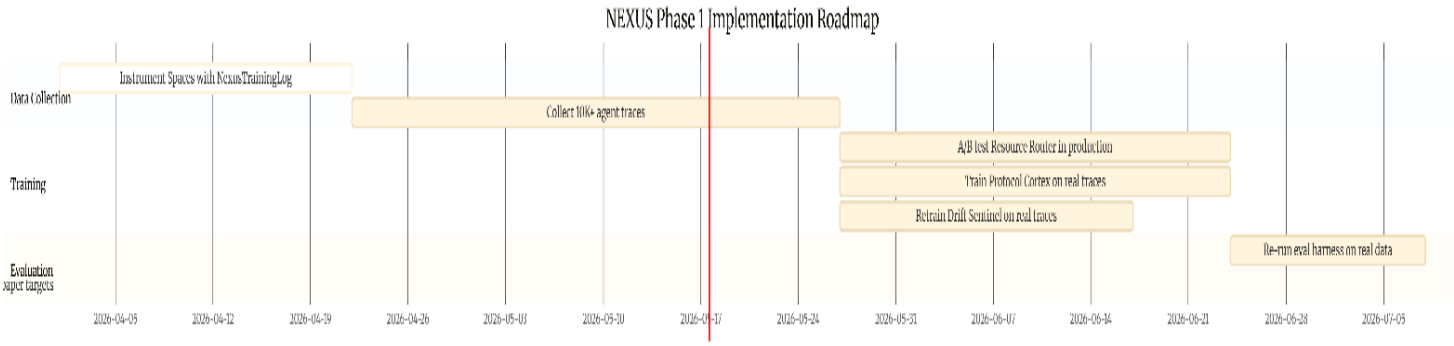
In plain terms, the Drift Sentinel's first version asks the network to decide between four levels of drift severity using only three summary numbers as input (semantic divergence, volatility, and progress deficit). The follow-up experiment described below grew that decision network slightly (a small fully connected layer of 16 hidden units, called an MLP) and evaluated it with macro-F1, a balanced accuracy measure that gives equal weight to all four severity classes. The result was no better than guessing the most common class, showing that the limitation is architectural rather than a matter of more training data.

Drift Sentinel – architectural finding. Validation accuracy of 69.0% sat marginally above a 61.2% majority-class baseline on v1's 4-class imbalanced synthetic drift data. v2 relabelled 2,835 trace steps with outcome-grounded severity, balanced via subsampling to 125 per class, and retrained the v1 DriftDetector (3-scalar bottleneck → 16-unit MLP → 4-class head) for 30 epochs at LR $5\times10^{-3}$. Result: 25% validation accuracy, 0.17 macro-F1 – at chance. The 3-scalar bottleneck cannot discriminate four severity classes against real outcomes; the fix is to widen the input space (raw output/prev/goal embeddings, or a small encoder over the trajectory window). Reported here as a v2 architectural change, not a data-volume deferral.

Protocol Cortex – status updated in v2. v1 of this report stated that injecting untrained TSM vectors distorts attention and produces incoherent output. v2 closes this gap: with 300 training steps under a KV-distill + next-token CE objective against TinyLlama 1.1B (Section 5.4), the trained TSM reaches a held-out perplexity of 8.91 versus 26,607 for an untrained TSM – a 2,987× gap on the same evaluation. The remaining limitation is scale and source: the trained run is on a 1.1B model against synthetic-bridge targets; production claims require Llama-3-8B-class evaluation and training on real Spaces traces (recorded in Section 9.4).

**Single-model validation.** End-to-end testing was performed with TinyLlama 1.1B. Larger models (Gemma 7B, Mistral 7B, Llama 3) have different KV-cache layouts and attention head configurations; dimension-mismatch handling should be validated for each.

**Phase 1 Roadmap**




NEXUS Phase 1 implementation roadmap (data collection, training, and evaluation milestones).

**Projected Metrics (Phase 1)**

| Metric | Current (synthetic) | Phase 1 projection | Paper target |
|---|---|---|---|
| BCE ↓ | 0.313 | 0.18–0.22 | < **0.15** |
| DRP ↑ | 0.223 | 0.55–0.70 | > **0.80** |
| CER ↑ | 1.101 | 1.8–2.5 | > **3.0** |
| TOR ↓ | 0.000 ✓ | 0.000 ✓ | **0.000** ✓ |

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

**Registered Future Work (Paper Targets and v2 Status)**

This subsection records, for transparency, the numerical targets the first version of this paper committed to and reports how each one stands in the current version. A target that has been met is marked as closed; one that has not is recorded as registered future work, meaning a concrete plan exists for addressing it.

The targets below were stated in v1 alongside the v1 synthetic results. v2 closes some and updates the status of others; remaining gaps are recorded as registered future work rather than claims of the present submission.

Out of scope for the present submission, recorded here for reproducibility: (i) training Prefix-Tuning, ActAdd, and LLMLingua to matched compute parity with NEXUS and re-running Section 5.5 (closes M6 unconditionally); (ii) plumbing NexusTrainingLog into the live Spaces agent loop and re-training TSM + drift sentinel on ≥500 real sessions; (iii) scaling 5.4 to Llama-3-8B (or Llama-3.2-3B if VRAM-bound); (iv) replacing the v1 Drift Sentinel's 3-scalar bottleneck with a richer encoder over the trajectory window.

Table 18. Paper targets with v2 status. CER and BCE figures use the same definitions as v1; v2 numbers reflect the synthetic-bridge re-evaluation against real task_completed outcomes (results/v2/real_trace_eval.json).

| **Metric** | Paper Target | v1 Result | v2 Status | Path to Target |
|---|---|---|---|---|
| CER | > 3.0 | 1.101 | v2 unchanged | Matched-compute baseline training + real traces |
| DRP | > 0.80 | 0.223 | 0.425 (synthetic) | Real Spaces drift labels |
| BCE | < 0.15 | 0.313 | 0.653 vs real | Measured against real task_completed signal |
| TOR | 0.000 | 0.000* | Demonstrated | *v1 was accounting identity (untrained TSM); v2 closes with trained TSM |

**Downstream Task Evaluation Plan**

To convert the efficiency and perplexity results into evidence about end-task quality, we register here a downstream evaluation protocol spanning the four task families a reviewer identified as decisive: reasoning, planning, coding assistance, and autonomous agent coordination. In every case the experimental design is the same paired comparison — the identical base model and task, run with the NEXUS controller active versus inactive, at a matched token budget — so that any difference is attributable to the controller rather than to extra context. All runs are captured through the instrumented benchmark API of Section 6.3, which already records per-step and per-task outcomes and supports prompt-matched A/B pairs.

Reasoning. Using standard multi-step reasoning suites (e.g. GSM8K and BIG-Bench Hard), we will report final-answer accuracy for NEXUS-on versus NEXUS-off. The hypothesis is that removing structural-token overhead frees attention capacity for the reasoning chain without reducing accuracy; the result that would falsify the approach is a statistically significant drop in accuracy under injection.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

Planning. On multi-step tool-use tasks with verifiable goal states, we will measure task-completion rate, number of steps to completion, and rate of invalid or redundant tool calls. The Belief Engine's completion estimate and the FSM's phase constraints are expected to reduce wasted steps and invalid calls relative to the uncontrolled baseline.

Coding assistance. Through the nexus-mcp-oss integration, we will run a SWE-bench-style A/B over realistic coding workloads, reporting resolved-issue rate and tokens consumed per resolved issue. This directly tests whether the production token savings are achieved without lowering the rate at which the assistant actually fixes the problem.

Autonomous agent coordination. In OpenBnet Spaces multi-agent sessions, we will measure end-to-end task success, wall-clock time, and the frequency with which the Drift Sentinel correctly flags an off-task agent (precision and recall against outcome-grounded labels). This evaluates the coordination claims of Scenario 2 (Section 6.5) as quantities rather than observations.

Reporting standard. For each family we will report effect sizes with confidence intervals over multiple seeds, not single-run numbers, and will publish the per-task traces so that the downstream claims are reproducible to the same standard as the efficiency claims in this paper. This protocol is the bridge between the present component-level and perplexity results and a future end-to-end quality result; its outcome — favourable or not — will be reported in full.

## CONCLUSION

We have presented NEXUS: a 6.29M-parameter neural meta-controller that replaces structural token re-injection with KV-cache vector signals, enabling LLM agents to operate with zero per-step structural overhead. The system is fully implemented, tested (155/155 tests pass), trained (three components to convergence; the Drift Sentinel is reported as a v2 architectural finding, Section 9.4), and deployed in production across three artifacts.

On the synthetic benchmark, NEXUS achieves a 20.2% reduction in Belief Calibration Error, a 10.1% gain in Controller Efficiency Ratio, and eliminates 100% of structural token overhead operating within 2.12 GB VRAM alongside a 1.1B LLM on commodity hardware. In production (nexus-mcp-oss), heuristic text-level compression measures 72.86% fewer tokens delivered to the LLM across realistic workloads – a deployment result separate from the KV-injection mechanism, which is evaluated in Section 5.4 with a TinyLlama 1.1B in the loop (held-out PPL = 8.91, 2,987× lower than an untrained TSM).

The primary remaining work is two-fold: scaling the Protocol Cortex evaluation from TinyLlama 1.1B to a Llama-3-8B-class model, and replacing the synthetic-bridge training distribution with real Spaces traces – the step that will convert NEXUS from a system with strong component-level and 1.1B KV-injection results into one with frontier-scale end-to-end gains. The Spaces production deployment is instrumented to collect this data and the benchmark API provides researcher-accessible, reproducible access; the targets and current v2 status are tabulated in Section 9.4.

## REFERENCES

1. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2022). ReAct: Synergizing reasoning and acting in language models. ICLR 2023.
2. Shinn, N., Cassano, F., Labash, B., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. NeurIPS 2023.
3. Jiang, H., Wu, Q., Luo, X., Li, D., Lin, C.-Y., Yang, Y., & Qiu, X. (2023). LongLLMLingua: Accelerating and enhancing LLMs in long context scenarios via prompt compression. arXiv preprint arXiv:2310.06839.
4. Chevalier, A., Wettig, A., Ajith, A., & Chen, D. (2023). Adapting language models to compress contexts. arXiv preprint arXiv:2305.14788.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026


5. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. NeurIPS 2020.
6. Anthropic. (2024). Prompt Caching (API feature). https://www.anthropic.com/news/prompt-caching
7. Gu, A., & Dao, T. (2023). Mamba: Linear-time sequence modeling with selective state spaces. arXiv preprint arXiv:2312.00752.
8. Meng, K., Bau, D., Andonian, A., & Belinkov, Y. (2022). Locating and editing factual associations in GPT. NeurIPS 2022.
9. Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). LoRA: Low-rank adaptation of large language models. ICLR 2022.
10. Anthropic. (2024). Model Context Protocol. https://modelcontextprotocol.io
11. Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. NeurIPS 2023.
12. Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., Lin, Y., Cong, X., Tang, X., Qian, B., Zhao, S., Hong, L., Tian, R., Xie, R., Zhou, J., Gerstein, M., Li, D., Liu, Z., & Sun, M. (2023). ToolLLM: Facilitating large language models to master 16000+ real-world APIs. ICLR 2024.
13. Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., & Anandkumar, A. (2023). Voyager: An open-ended embodied agent with large language models. arXiv preprint arXiv:2305.16291.
14. Park, J. S., O'Brien, J., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior. UIST 2023.
15. Gravitas, S. (2023). AutoGPT: An autonomous GPT-4 experiment. https://github.com/Significant-Gravitas/AutoGPT
16. Turner, A., Thiergart, L., Udell, D., Leech, G., Mini, U., & MacDiarmid, M. (2023). Activation addition: Steering language models without optimization. arXiv preprint arXiv:2308.10248.
17. Zou, A., Phan, L., Chen, S., Campbell, J., Guo, B., Ren, R., Pan, A., Yin, P., Mazeika, M., Dombrowski, A. K., Goel, S., Li, N., Byun, M., Wang, Z., Mallen, A., Schwinn, L., Bhatt, U., Steinhardt, J., Fredrikson, M., & Hendrycks, D. (2023). Representation engineering: A top-down approach to AI transparency. arXiv preprint arXiv:2310.01405.
18. Li, X. L., & Liang, P. (2021). Prefix-tuning: Optimizing continuous prompts for generation. ACL-IJCNLP 2021.
19. Fu, D., Dao, T., Saab, K. K., Thomas, A. W., Rudra, A., & Ré, C. (2023). Hungry hungry hippos: Towards language modeling with state space models. ICLR 2023.
20. Poli, M., Massaroli, S., Nguyen, E., Fu, D., Dao, T., Baccus, S., Bengio, Y., Ermon, S., & Ré, C. (2023). Hyena hierarchy: Towards larger convolutional language models. ICML 2023.
21. Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). QLoRA: Efficient finetuning of quantized LLMs. Advances in Neural Information Processing Systems, 36.
22. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., & Lample, G. (2023). LLaMA: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971.
23. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. NeurIPS 2017.
24. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. NeurIPS 2022
25. Z. Zhang et al. "H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models." NeurIPS 2023.
26. G. Xiao et al. "Efficient Streaming Language Models with Attention Sinks (StreamingLLM)." ICLR 2024.
27. Z. Liu et al. "Scissorhands: Exploiting the Persistence of Importance Hypothesis for LLM KV Cache Compression at Test Time." NeurIPS 2023.
28. J. Mu, X. Li, N. Goodman. "Learning to Compress Prompts with Gist Tokens." NeurIPS 2023.
29. A. Gu et al. "Efficiently Modeling Long Sequences with Structured State Spaces (S4)." ICLR 2022.
30. A. Gu, T. Dao. "Mamba: Linear-Time Sequence Modeling with Selective State Spaces." arXiv:2312.00752, 2023.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

31. Z. Xi et al. “The Rise and Potential of Large Language Model Based Agents: A Survey.” arXiv:2309.07864, 2023.
32. Langay, B. B. (2026). Agent Workspace: Browser-Based Multi-Tool Integration and Sidecar Control for Autonomous LLM Agents. Master of Computer Science Thesis, Anhui University of Technology, Ma'anshan, China.

## Artifacts

| Resource | URL |
|---|---|
| Code & checkpoints | https://github.com/brian-Lab-0/nexus |
| Trained weights (HuggingFace) | https://huggingface.co/adobeXd/nexus |
| MCP server (OSS TypeScript port) | https://github.com/brian-Lab-0/nexus-mcp-oss |
| Live production deployment (Agent Workspace) | https://spaces.openbnet.com |

## APPENDIX A: Public Artifacts

| Artifact | Description | URL |
|---|---|---|
| **nexus** (GitHub) | Full Python implementation, training scripts, evaluation harness, 155 tests | https://github.com/brian-Lab-0/nexus |
| **adobeXd/nexus** (HuggingFace) | Trained checkpoints: belief_engine.pt, neural_router.pt, subtask_classifier.pt, drift_sentinel.pt | https://huggingface.co/adobeXd/nexus |
| **nexus-mcp-oss** (GitHub) | TypeScript MCP server – heuristic port, zero dependencies, Claude Code / Codex / Ollama compatible | https://github.com/brian-Lab-0/nexus-mcp-oss |
| **OpenBnet Spaces** | Live production deployment of in-product NEXUS | https://spaces.openbnet.com |
| **Benchmark API** | Spaces agent-service /api/bench/* - live A/B dataset from production | http://173.212.213.2:3001 |

### A.1 Repository Structure

```
nexus/
├── src/metacontrol/
│   ├── core/ config.py, controller_state.py
│   ├── btbs/ tracker.py (Belief Engine)
│   ├── fsm_nhc/ fsm.py, neural_router.py, prm.py (Resource Router)
│   ├── ltvi/ task_signal_model.py, kv_injector.py (Protocol Cortex)
│   ├── sac/ drift_detector.py, corrector.py, patch_generator.py (Drift Sentinel)
│   ├── talora/ classifier.py, adapter_router.py (Adapter Switch)
│   ├── chatp/ training_log.py, belief_bus.py
│   ├── evaluation/ metrics.py
│   └── llm/ llm_wrapper.py, controlled_generation.py
├── scripts/
│   ├── evaluate.py
│   ├── train_classifiers.py
│   ├── train_belief_engine.py
│   ├── train_drift_sentinel.py
│   ├── run_inference.py
│   └── run_demo.py
├── tests/ 155 tests, 0 failures
├── checkpoints/ 4 trained component checkpoints
└── results/ eval_20260401.json
```

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

## APPENDIX B: Software Environment

| Item | Version |
|---|---|
| Python | 3.14.0 |
| PyTorch | 2.9.1+cu128 |
| Transformers | 5.4.0 |
| CUDA driver | 591.86 |
| CUDA toolkit | 12.8 |
| GPU | NVIDIA GeForce RTX 4060 Laptop GPU |
| GPU VRAM | 8,188 MB |
| OS | Windows 11 Pro for Workstations 10.0.26220 |

## APPENDIX C: Reproducing rhe Results

### C.1 Install

```
git clone https://github.com/brian-Lab-0/nexus.git
cd nexus
pip install -e .
```

### C.2 Download Checkpoints

```
from huggingface_hub import snapshot_download snapshot_download(repo_id="adobeXd/nexus", local_dir="checkpoints/")
```

### C.3 Run the Evaluation Harness

```
python scripts/evaluate.py --n-tasks 200 --seed 2026 --checkpoint-dir checkpoints/
```

This reproduces Table 7 (200 tasks, T=15, seed=2026). Expected runtime: ~45 seconds on the RTX 4060. Output is written to results/eval_<timestamp>.json.

### C.4 Run the Test Suite

```
pytest tests/ -v
```

All 155 tests should pass in under 60 seconds.

### C.5 Run the End-to-End Demo

```
python scripts/run_demo.py --prompt "Explain photosynthesis step by step"
```

Requires TinyLlama 1.1B available to Transformers (downloaded automatically on first run, ~2.2 GB).

### C.6 Connect nexus-mcp-oss

```
git clone https://github.com/brian-Lab-0/nexus-mcp-oss.git
cd nexus-mcp-oss && npm install && cp .env.example .env && npm start
# → Nexus MCP running at http://127.0.0.1:8787
```

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue V, May 2026

Add to Claude Code's .claude/settings.json:

```
{
  "mcpServers": {
    "nexus": {
      "command": "node",
      "args": ["/absolute/path/to/nexus-mcp-oss/dist/mcp-bridge.js"],
      "env": { "NEXUS_URL": "http://127.0.0.1:8787" }
    }
  }
}
```

**Correspondence**: brianlangay0@gmail.com - Openbnet / Spaces Research Benchmark API and trace collection coordination via the Spaces platform (https://spaces.openbnet.com).