Research Article

MiND: A Minimal Deterministic Middleware for Auditable LLM Interaction

Edervaldo José de Souza Melo[1]

1. Independent researcher

Large language models are increasingly used in workflows that require inspectability, traceability, and reproducible interaction procedures. However, direct use of monolithic chat interfaces often obscures how system-level instructions are applied, how requests are composed, and how interaction records can be retrieved for later inspection. This paper presents MiND, a minimal deterministic middleware for auditable LLM interaction. MiND does not attempt to make the underlying language model deterministic in a strict output sense; instead, it makes the middleware layer deterministic and inspectable by externalizing cycle configuration, explicitly composing system-level instructions, exposing a lightweight HTTP interface, and recording cycle-level data including input, configuration, output, and metadata. A reference implementation built with FastAPI provides endpoints for health checking, configuration inspection, chat execution, and retrieval of the latest interaction cycle. We argue that this design offers a low-friction approach to auditability, traceability, and reproducibility support in LLM-mediated workflows, while remaining lightweight enough to serve as a minimal executable reference for future modular extensions.

Corresponding author: Edervaldo José de Souza Melo, edersouzamelo@gmail.com

1. Introduction

Large language models (LLMs) have become a common computational layer in research, prototyping, and decision-support workflows. Built on Transformer-based architectures [1], these systems are increasingly discussed within the broader landscape of foundation models and their infrastructural implications [2]. They are used not only for one-off prompting, but also for repeated interaction cycles

involving instruction layering, contextual memory, iterative refinement, and task-specific behavioral constraints.

In many current deployments, user inputs are sent directly to model APIs or platform-managed chat environments. Although this interaction model is convenient, it often obscures the procedural structure of each invocation. System-level instructions, runtime parameters, context assembly, and interaction history may remain partially hidden, externally managed, or inconsistently documented. As a result, it becomes difficult to determine what exact configuration was active in a given interaction, what contextual frame preceded a response, and how a particular output can be inspected after the fact.

MiND was designed to address this practical gap at the middleware level.[1] Rather than attempting to alter the internal statistical behavior of a language model, MiND makes the surrounding interaction pipeline explicit, lightweight, and inspectable. The system externalizes cycle configuration, composes system-level instructions in a controlled manner, exposes a minimal HTTP interface, and records cycle-level artifacts including input, configuration, output, and execution metadata. In this sense, MiND does not make the model itself deterministic in a strict output sense; instead, it makes the request-handling layer deterministic and recoverable. The name MiND originally derives from “Minimal Nemosine Design”; in the present paper, it is used in its software-publication sense as a Minimal Deterministic Middleware for auditable LLM interaction.

This distinction is central. The underlying LLM remains probabilistic, provider-dependent, and semantically generative. MiND does not claim to infer model internals, eliminate uncertainty, or guarantee semantic reproducibility of responses. Its contribution lies elsewhere: each interaction can be treated as an explicit computational event with a recoverable structure. This allows the interaction pipeline to be inspected as a sequence of defined operations rather than as an opaque conversational side effect.

The reference implementation examined in this paper adopts a minimal architecture centered on four operational capabilities: service health inspection, external configuration exposure, chat execution, and retrieval of the most recent recorded cycle. Empirically, this enables a lightweight yet concrete form of auditability. A local execution can reveal whether the service is running, what cycle configuration is active, what input was submitted, what output was returned, and what metadata were associated with the interaction. Such functionality is modest by design, but sufficient to support a minimal executable reference for auditable LLM mediation.

The broader motivation for MiND is infrastructural rather than model-centric. The system is intended for settings in which users or researchers need a thin orchestration layer between themselves and an LLM provider, allowing them to retain procedural visibility over how interactions are constructed and logged. This is especially relevant when interaction traces need to be revisited, compared, or documented across sessions, without relying exclusively on platform-native histories or hidden conversation state.

Accordingly, this paper presents MiND as a minimal deterministic middleware for auditable LLM interaction. The aim is not to propose a complete agent architecture, nor to claim strong guarantees beyond what the implementation demonstrates. Instead, the paper introduces a small but operationally meaningful software layer that supports externalized configuration, cycle-level logging, and post-hoc inspection of recent interactions.

This concern with explicit interaction structure is consistent with broader discussions about documentation, reproducibility, and the limits of opaque language-model deployment in research and applied settings [3], [4].

1.1. Contributions of this Work

This paper makes four main contributions. First, it defines MiND as a deterministic middleware layer whose determinism applies to the orchestration and logging procedure rather than to the LLM output itself. Second, it describes a minimal reference implementation that exposes configuration, execution, and cycle-retrieval capabilities through a lightweight HTTP interface. Third, it demonstrates that auditable interaction can be operationalized through recoverable cycle artifacts containing input, configuration, output, and metadata. Fourth, it positions MiND as a minimal executable reference for future modular extensions without overstating the maturity or scope of the current implementation.

2. Design Rationale

MiND was designed under a deliberately narrow engineering premise: when direct interaction with an LLM becomes operationally convenient but procedurally opaque, a minimal middleware layer can restore inspectability without requiring changes to the underlying model. The system therefore prioritizes explicit mediation over internal modification. Its core rationale is not to compete with model architectures, agent frameworks, or full orchestration platforms, but to establish a small and operationally meaningful layer in which interaction procedures can be made visible, recoverable, and externally configurable.

This rationale is grounded in a distinction between model behavior and interaction behavior. The former remains largely controlled by the LLM provider, the model family, runtime sampling parameters, and the semantic variability of the prompt itself. The latter, however, can be structured by software. MiND acts precisely at this second level. It does not determine what the model must mean, but it can determine how a request is assembled, what configuration governs the cycle, what metadata are recorded, and how the most recent interaction can be retrieved for inspection.

From this perspective, the term *deterministic* in MiND refers to the middleware procedure rather than to the semantic output. Given the same software state and the same external configuration path, the request-handling pipeline follows an explicit sequence: load cycle configuration, compose the request frame, execute the model call, persist cycle-level data, and expose the resulting record through retrieval endpoints. This procedural determinism is modest but useful. It creates a stable object of inspection around an otherwise probabilistic model invocation.

A second design principle is minimality. MiND intentionally avoids embedding a large internal ontology of roles, complex routing hierarchies, autonomous agent loops, or hidden decision layers. Although such extensions may be possible in future work, the present implementation focuses on the smallest architecture capable of demonstrating auditable mediation. This minimality serves two purposes. First, it reduces the conceptual distance between the implementation and the claims made in the paper. Second, it makes the system easier to inspect, reproduce, and adapt in settings where researchers or developers need a thin control layer rather than a full orchestration ecosystem.

A third principle is externalized configuration. Instead of hard-coding the entire interaction logic inside opaque application state, MiND exposes cycle-relevant parameters through a configuration object that can be inspected independently of a single interaction. In the reference implementation, this includes the active mode, model identifier, runtime parameters, and the system-level template used to frame the request. Externalization matters because it separates the *interaction event* from the *interaction policy*. This separation improves clarity when inspecting what happened in a cycle and why it was processed under a particular framing condition.

A fourth principle is cycle-level recoverability. MiND treats each invocation as a structured cycle rather than as a transient conversational event. The practical consequence is that an interaction can be revisited as a bounded record containing input, active configuration, generated output, and execution metadata. This approach supports post-hoc inspection without requiring continuous access to hidden platform histories or undocumented interface state. In the current implementation, recoverability is intentionally

limited to the latest cycle, which is sufficient for demonstrating the basic concept while preserving architectural simplicity.

A fifth principle is non-parametric steering. MiND does not retrain, fine-tune, or otherwise modify model weights. Instead, it operates through explicit request framing at middleware level. In the reference implementation, this is achieved through a configurable system template combined with user input and runtime parameters. The resulting behavior should not be interpreted as strong control over model semantics, but rather as a disciplined and inspectable method for shaping the conditions under which the model is called. This is why MiND is better understood as an infrastructural intervention than as a modeling innovation.

Taken together, these principles define the practical niche of MiND. The system is intended for contexts in which auditability, traceability, and procedural clarity matter more than architectural breadth. Its value lies not in maximizing automation, but in making each interaction cycle easier to inspect as a software-mediated event. For this reason, MiND is presented here as a minimal executable reference: small enough to remain honest about its current scope, yet structured enough to support future modular growth without changing its foundational rationale.

3. Architecture

MiND adopts a thin middleware architecture positioned between a user-facing client and an external LLM provider. The purpose of this layer is not to replace the provider-facing API, but to mediate the interaction through a minimal and inspectable execution path. In the reference implementation, the architecture is intentionally compact and can be understood as four functional surfaces: service inspection, configuration exposure, chat execution, and cycle retrieval.

At a high level, the architecture begins with a user request submitted to the middleware through a lightweight HTTP interface. The middleware then loads the active interaction configuration, composes the request frame to be sent to the LLM provider, executes the model call, records a structured cycle artifact, and returns the generated output to the client. The same middleware also exposes endpoints that allow the current configuration and the latest recorded cycle to be inspected independently of the chat execution itself. This separation between execution and inspection is one of the central architectural properties of MiND.

3.1. High-Level Components

The reference implementation can be described in terms of five logical components:

1. **HTTP service layer**: exposes the middleware through documented endpoints, enabling both execution and inspection.
2. **Configuration layer**: stores and exposes the active cycle configuration, including operational mode, model identifier, runtime parameters, and the system-level template used to frame requests.
3. **Execution layer**: receives user input, combines it with the active configuration, and performs the provider-facing LLM call.
4. **Cycle recording layer**: persists a structured representation of the interaction cycle, including input, configuration snapshot, output, and metadata.
5. **Retrieval layer**: exposes the most recent recorded cycle as an inspectable artifact.

These components should not be interpreted as a claim of deep modular sophistication. Their relevance lies in the fact that each one corresponds to an explicit operational function that can be inspected or tested separately. The architecture is therefore modular in a minimal software-engineering sense: distinct concerns are separated into inspectable middleware responsibilities, even though the implementation remains deliberately lightweight.

3.2. Request Handling Cycle

The core interaction path of MiND follows a simple and explicit sequence:

1. receive user input through the chat endpoint;
2. load the active cycle configuration;
3. compose the request frame, including the configured system-level template;
4. send the composed request to the LLM provider;
5. receive the provider output;
6. create a cycle record with a unique identifier;
7. persist the cycle artifact together with execution metadata;
8. return the generated output to the client.

This sequence is the main locus of determinism claimed by MiND. The semantic content of the provider output may vary, since the underlying model remains probabilistic. However, the middleware procedure

governing how the interaction is assembled, executed, recorded, and exposed for later retrieval follows a fixed and inspectable structure. In this sense, MiND stabilizes the interaction pipeline even when it cannot stabilize the model output itself.

3.3. Exposed Endpoints

The current implementation exposes a minimal set of endpoints aligned with the architectural rationale of the system:

- `/health`: returns service status and runtime version information;
- `/ame/config`: exposes the active interaction configuration;
- `/chat`: executes a chat cycle by sending configured input to the provider;
- `/ame/last`: retrieves the most recently recorded cycle artifact.

Together, these endpoints define a minimal inspectable middleware surface. The architecture therefore does not rely exclusively on successful model invocation. Even when the chat endpoint is not used, the system can still expose operational state through health and configuration inspection. Conversely, when a chat interaction occurs, the resulting cycle can be inspected post hoc through the retrieval endpoint.

3.4. Cycle Artifact Structure

A central architectural object in MiND is the cycle artifact. Rather than treating an LLM interaction as an ephemeral message exchange, the middleware represents it as a bounded record containing the elements necessary for later inspection. In the current implementation, this structure includes at least four conceptual fields:

1. **input**: the user-submitted text;
2. **config**: the active configuration snapshot used for that cycle;
3. **output**: the model-generated reply returned through the middleware;
4. **meta**: execution metadata such as timestamps, latency, and provider-related information.

In addition, each cycle is associated with a unique `cycle_id`, which functions as a stable handle for identifying the interaction instance. This data structure is architecturally important because it transforms a conversational event into a software object that can be logged, inspected, and compared.

3.5. Architectural Scope

The present architecture should be understood as a minimal executable reference rather than as a full orchestration framework. It does not currently implement advanced agent routing, long-horizon memory management, multi-cycle planning, or strong internal role specialization. These features may be compatible with the architectural rationale of MiND in future versions, but they are not required for the present contribution. What matters here is that the implemented architecture already demonstrates the core idea of explicit middleware mediation for auditable LLM interaction.

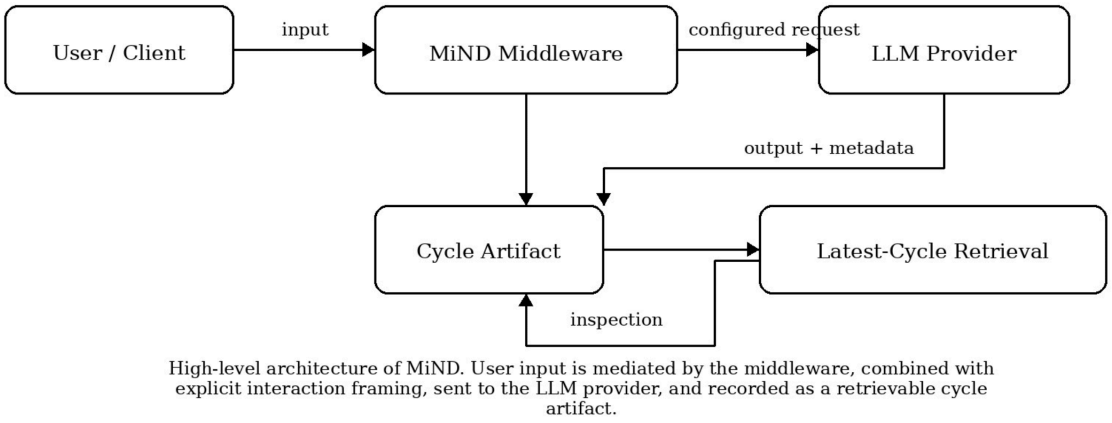


Figure 1. High-level architecture of MiND

4. Reference Implementation

The current MiND implementation was developed as a lightweight proof-of-operation rather than as a production orchestration framework. Its role is to instantiate the architectural rationale of the system in executable form, showing that auditable LLM mediation can be operationalized through a minimal middleware layer with externally inspectable state. The implementation is intentionally compact and prioritizes clarity of function over breadth of features.

Technically, the reference implementation exposes its interface through FastAPI, which provides a simple HTTP surface and automatic interactive documentation. This choice is consistent with the overall design philosophy of MiND: the middleware should remain easy to run locally, easy to inspect during execution, and easy to validate without requiring a complex deployment stack. The implementation therefore favors

a minimal developer-facing runtime that can be launched locally and inspected through documented endpoints.

In operational terms, the implementation supports four core capabilities. First, it exposes a health endpoint that reports service availability and runtime version information. Second, it exposes the active interaction configuration through a dedicated configuration endpoint. Third, it executes provider-facing chat calls through a chat endpoint that receives user input and returns generated output. Fourth, it exposes the most recently recorded interaction cycle through a retrieval endpoint. These four capabilities are sufficient to demonstrate that the middleware does not merely wrap the model call, but structures it as an inspectable software event.

The current implementation also makes explicit use of a configurable system-level template in the request composition process. This is important because it demonstrates that MiND is not limited to raw pass-through interaction. Instead, the middleware can inject a stable framing condition at cycle level while still keeping that framing externally visible through configuration inspection. In this sense, the implementation already supports a minimal form of non-parametric behavioral steering, although such steering remains limited to explicit middleware framing rather than any internal model modification.

Another relevant property of the implementation is that it materializes the notion of a cycle artifact in concrete retrievable form. When a chat interaction is executed, the resulting record contains a unique cycle identifier together with the submitted input, the active configuration snapshot, the returned output, and execution metadata. This is a key implementation feature because it converts an otherwise transient interaction into a bounded software object. The retrievable cycle artifact is what makes auditability operational rather than merely conceptual in the present system.

It is important to state the limitations of the current implementation clearly. The system does not yet provide advanced routing across multiple specialized modules, long-term memory management, provider-agnostic abstraction at scale, or formal verification guarantees. Likewise, the current retrieval mechanism is intentionally narrow, focusing on the latest recorded cycle rather than on a broader indexed history. These limitations are not incidental omissions but reflections of the minimal scope of the present version. The implementation is intended to demonstrate the viability of the middleware concept without overstating its maturity.

For this reason, the present reference implementation should be understood as a minimal executable reference rather than as a finished product. Its technical significance lies in the fact that the central claims of the paper can be directly tied to observable runtime behavior: the service can be launched

locally, the active configuration can be inspected, a chat cycle can be executed, and the resulting cycle record can be retrieved. This alignment between design claim and executable artifact is one of the main strengths of MiND in its current form.

5. Operational Demonstration

To examine whether MiND's core claims are supported by observable runtime behavior, the reference implementation was executed locally and inspected through its documented HTTP interface. The purpose of this demonstration is not to benchmark model quality or compare providers, but to verify that the middleware exposes the minimal operational properties claimed in this paper: service availability, configuration inspectability, chat-cycle execution, and retrieval of a structured cycle artifact.

5.1. Local Execution Context

The implementation was launched as a local FastAPI service and inspected through the automatically generated interactive documentation interface. This setup is appropriate for the present contribution because MiND is introduced as a lightweight executable reference rather than as a production deployment. The demonstration therefore focuses on whether the middleware can be run, queried, and inspected in a transparent way by a local user or researcher.

5.2. Observed Endpoint Behavior

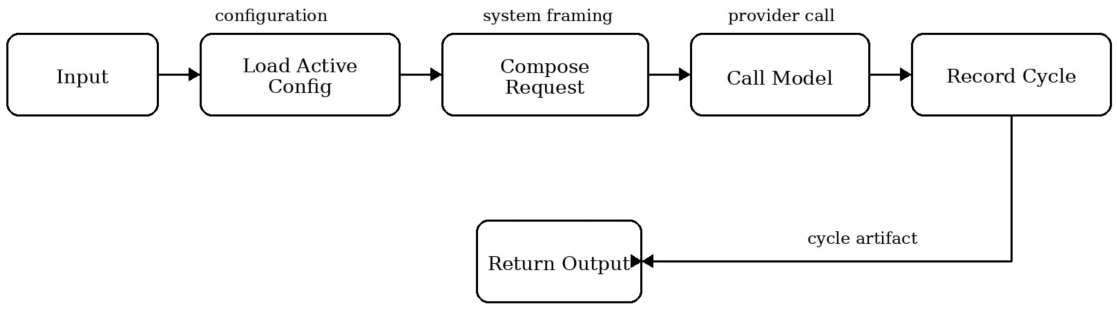
Four endpoints were inspected during the local execution.

First, the `/health` endpoint returned a successful response indicating that the service was active and exposing runtime version information. In the observed execution, the service reported `"ok": true` together with version metadata. This confirms that MiND exposes a minimal runtime introspection surface independent of any model invocation.

Second, the `/ame/config` endpoint returned the active interaction configuration. The observed response exposed a configuration object containing at least the active mode, the selected model identifier, runtime parameters such as temperature and token limits, and a system-level template used in request composition. This is operationally significant because it shows that interaction framing is not hidden in undocumented application state; it can be inspected as part of the middleware's explicit configuration surface.

Third, the `/chat` endpoint successfully executed a chat cycle. In the observed test, a simple textual input was submitted and the middleware returned a generated reply together with a unique `cycle_id`. This is a critical result for the present paper because it confirms that MiND does not merely expose configuration passively: it performs an actual provider-facing interaction while preserving a retrievable handle for the resulting cycle.

Fourth, the `/ame/last` endpoint returned the most recently recorded cycle artifact. The observed cycle included the submitted input, the active configuration snapshot used during execution, the model-generated output, and execution metadata such as timestamp and latency. This endpoint is the strongest operational support for the paper's main claim, because it demonstrates that a completed interaction can be revisited as a bounded and inspectable software object.



Request-handling cycle in MiND. User input is processed through a fixed middleware sequence in which the active configuration is loaded, the request is composed, the model call is executed, the resulting interaction is recorded, and the output is returned.

Figure 2. Request-handling cycle in MiND

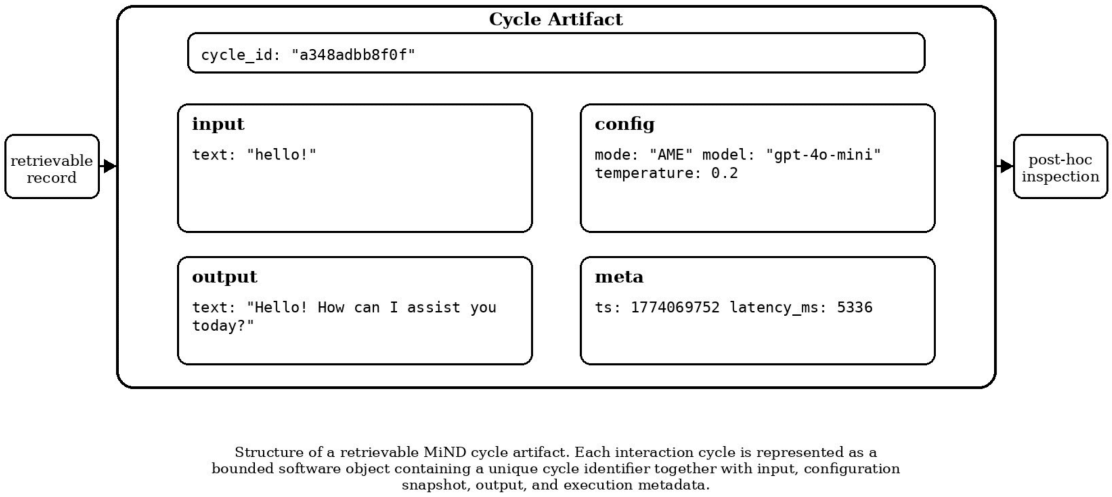
5.3. What the Demonstration Establishes

Taken together, the local execution demonstrates five concrete properties of the current MiND implementation.

1. **Runnable middleware layer**: the service can be started locally and inspected through a documented HTTP interface.
2. **Runtime introspection**: the middleware exposes service health and version state independently of chat execution.
3. **Inspectable interaction policy**: the active cycle configuration can be queried explicitly, including framing-related parameters.

4. **Executable interaction cycle**: a user input can be processed through the middleware and sent to the LLM provider, producing a generated reply and a unique cycle identifier.
5. **Recoverable cycle artifact**: the most recent interaction can be retrieved with structured fields covering input, configuration, output, and metadata.

These observations are sufficient to support the argument that MiND operationalizes auditable interaction at the middleware level. The demonstration does not prove semantic determinism of model output, nor does it establish provider-independent reproducibility in a strict sense. What it does show is that the software layer surrounding the model call has been made explicit enough to support post-hoc inspection of how an interaction cycle was framed and executed.



Structure of a retrievable MiND cycle artifact. Each interaction cycle is represented as a bounded software object containing a unique cycle identifier together with input, configuration snapshot, output, and execution metadata.

Figure 3. Structure of a retrievable MiND cycle artifact

5.4. Interpretive Significance

The importance of this demonstration lies in the distinction between conversational opacity and procedural inspectability. In many ordinary LLM usage scenarios, an interaction may yield a useful response without preserving a clearly inspectable record of how the interaction was configured at runtime. In MiND, by contrast, the middleware creates a visible procedural envelope around the interaction. The returned `cycle_id`, the exposed configuration, and the retrievable cycle artifact together provide a minimal but concrete basis for treating LLM interaction as an auditable computational event.

From an engineering standpoint, this is enough to justify MiND as a minimal executable reference. The system need not implement broad orchestration features in order to be meaningful. Its present value lies in showing that a thin software layer can already improve the inspectability of LLM-mediated interaction by externalizing configuration and by preserving cycle-bounded records that can be queried after execution.

5.5. Scope of the Demonstration

The demonstration reported here is intentionally narrow. It does not evaluate response quality, adversarial robustness, scalability, multi-user concurrency, or long-term storage behavior. It also does not test complex routing policies or modular specialization, since those are outside the scope of the current implementation. The goal is simply to verify that the minimal claims advanced in this paper correspond to directly observable system behavior. On that criterion, the implementation succeeds: MiND can be run locally, queried through explicit endpoints, used to execute a chat cycle, and inspected through retrievable cycle data.

6. Discussion and Limitations

MiND should be interpreted as an infrastructural contribution rather than as a claim about model internals. Its central value lies in making the interaction layer around an LLM more explicit, inspectable, and recoverable. In this sense, the system contributes to procedural clarity, not to semantic certainty. The reference implementation shows that a thin middleware layer can externalize configuration, structure the request path, and preserve a retrievable interaction record without requiring retraining, fine-tuning, or architectural modification of the underlying model.

This contribution is modest by design. MiND does not solve interpretability in the strong sense, and it does not reveal how the LLM internally produces a given response. What it offers is a more transparent software envelope around the interaction. That envelope is useful in contexts where the practical problem is not understanding the model's internal weights, but understanding how a particular request was framed, executed, and recorded at middleware level.

A related point concerns reproducibility, a concern that has been repeatedly emphasized in machine learning research as an infrastructural rather than purely experimental issue [4]. MiND supports reproducibility in a bounded procedural sense: it preserves the input, the active configuration snapshot, and execution metadata associated with a cycle. This can help users or researchers revisit how an

interaction was conducted and under what middleware conditions it occurred. However, such support should not be confused with strict semantic reproducibility of outputs. Because the underlying LLM remains probabilistic and provider-dependent, re-executing a similar cycle may still yield different responses even when the middleware procedure is kept stable. For this reason, MiND is better described as supporting auditable and traceable interaction than as guaranteeing exact response replication.

The notion of determinism used throughout this paper must therefore remain carefully delimited. MiND does not impose determinism on the generative model itself. Instead, it imposes determinism on the surrounding request-handling procedure: which configuration is loaded, how the request frame is composed, what metadata are recorded, and how the resulting cycle can be retrieved. This is a weaker notion than deterministic inference at model level, but it is also a more defensible and operationally relevant one for the current implementation.

The present system is also limited in architectural scope. It does not yet implement indexed long-term cycle history, advanced routing across multiple specialized modules, multi-agent coordination, formal policy verification, or provider-agnostic abstraction at a mature level. Its retrieval mechanism is narrow and intentionally focused on the latest cycle. Likewise, the current implementation should not be read as evidence of scalability under concurrent workloads or as a hardened production system. These absences are important because they define the actual scope of the contribution: MiND is a minimal executable reference, not a comprehensive orchestration platform.

Another limitation concerns framing specificity. In the observed implementation, the active system-level template is configurable and externally inspectable, which is a strength of the middleware design. At the same time, the specific template used during a given execution may reflect project-specific assumptions or domain language. This means that the current implementation demonstrates configurability more clearly than general neutrality. Future iterations should further separate middleware logic from domain-specific framing conventions in order to strengthen portability across contexts.

A further limitation is evaluative. The present paper does not provide comparative experiments against alternative middleware systems, direct API usage, or platform-native chat interfaces. It also does not evaluate response quality, safety performance, adversarial robustness, or user-study outcomes. This omission is deliberate: the current goal is to establish the viability of the software concept and to document its minimal operational properties. Comparative and large-scale evaluation remain appropriate directions for subsequent work once the software layer itself has been stabilized further.

Despite these limitations, MiND occupies a meaningful technical niche. Many LLM usage scenarios do not initially require full agent architectures or complex orchestration ecosystems; they require a minimal layer that makes interaction policy visible and recent cycles recoverable. In such settings, the present implementation already provides value by converting an otherwise opaque interaction into an inspectable middleware event. The broader significance of MiND is therefore not that it solves every orchestration problem, but that it demonstrates how little software is needed to begin making LLM interaction more auditable in practice.

Future work can extend MiND along several directions while preserving its minimal core rationale. These include broader cycle indexing, modular routing policies, more explicit provider abstraction, stronger packaging consistency, neutralized default templates, and systematic evaluation across repeated interaction settings. The challenge for future versions will be to expand functionality without sacrificing the main virtue of the current system: explicitness of procedure.

7. Conclusion

This paper introduced MiND as a minimal deterministic middleware for auditable LLM interaction. The system was presented not as a solution to model interpretability in the strong sense, but as a lightweight infrastructural layer that makes the interaction procedure around an LLM more explicit, inspectable, and recoverable. By externalizing cycle configuration, structuring request composition, exposing a simple HTTP interface, and preserving cycle-level artifacts, MiND turns a model invocation into a bounded software event that can be revisited after execution.

The reference implementation demonstrates that this concept is operationally viable in minimal form. A local execution showed that the middleware can be launched, queried for service health, inspected for active configuration, used to execute a chat cycle, and then queried again to retrieve the resulting cycle artifact. These properties are sufficient to support the paper's central claim: MiND does not make the underlying model deterministic, but it does make the middleware procedure deterministic and auditable in a practically meaningful sense.

The significance of MiND lies in the fact that it addresses a real but often overlooked layer of LLM use. Between direct prompting and full orchestration frameworks, there is a technical space for small systems that improve procedural visibility without demanding large architectural commitments. MiND occupies that space. Its present contribution is modest but concrete: it shows that a thin middleware layer can already improve traceability and post-hoc inspection of LLM-mediated interaction.

At the same time, the paper has deliberately maintained a narrow scope. MiND is not presented here as a mature multi-agent framework, a complete provider abstraction layer, or a mechanism for guaranteeing semantic reproducibility. Its current value lies in its explicitness, executability, and conceptual restraint. These qualities make it suitable as a minimal executable reference for future work on modular extensions, broader cycle management, and more systematic evaluation.

In that sense, MiND should be understood as a foundational software contribution: small in scale, but precise in purpose. Its core proposal is that auditable interaction with LLMs does not necessarily require heavy orchestration. In some cases, a minimal deterministic middleware is enough to make the interaction procedure visible, structured, and recoverable—and that alone can already be technically useful.

8. Availability

MiND is available as an open-source software artifact through its public source-code repository and as an installable Python package.

- **Source code repository:** `https://github.com/edersouzamelo/nemosine-10-MiND`
- **Python package (PyPI):** `https://pypi.org/project/nemosine-mind/`

The reference implementation described in this paper can be executed locally through its FastAPI-based service interface. At the time of writing, the system exposes a minimal documented HTTP surface for health inspection, configuration exposure, chat-cycle execution, and latest-cycle retrieval. The software should therefore be understood as a minimal executable reference implementation intended for inspection, experimentation, and future extension.

Because MiND is under active development, implementation details such as package versioning, default templates, and retrieval scope may evolve across releases. For this reason, the GitHub repository should be treated as the primary development source, while the PyPI package provides a lightweight installation channel for executable use.

The software artifact discussed in this paper is publicly available for inspection and reuse [5].

Footnotes

[1] MiND is developed within the broader Nemosine Nous research program, but the present paper is intentionally limited to its minimal technical role as an auditable LLM interaction middleware.

References

1. ^*Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez A, Kaiser, Polosukhin I (2017). "Attention Is All You Need."*
2. ^*Bommasani R, Hudson D, Adeli E, Altman R, Arora S, von Arx S, Bernstein M, Bohg J, Bosselut A, Brunskill E, others (2021). "On the Opportunities and Risks of Foundation Models." arXiv. arXiv:2108.07258.*
3. ^*Bender E, Gebru T, McMillan-Major A, Shmitchell S (2021). "On the Dangers of Stochastic Parrots: Can Lan guage Models Be Too Big?."*
4. a, b*Pineau J, Vincent-Lamarre P, Sinha K, Larivière V, Beygelzimer A, d'Alché-Buc F, Fox E, Larochelle H (202 1). "Improving Reproducibility in Machine Learning Research: A Report from the NeurIPS 2019 Reproducibil ity Program." J Mach Learn Res. 22(164):1–20.*
5. ^*Melo E (2026). "MiND: A Minimal Deterministic Middleware for Auditable LLM Interaction." Software rep ository.*

Declarations

Funding: No specific funding was received for this work.

Potential competing interests: No potential competing interests to declare.