SENTRY-LOGIC Phase 1 Technical Plan

System Goals:

* Observe prompt → response cycles without accessing weights.

* Generate logs using predefined symbols (e.g., Δ, Ω, Λ, ⇄).

* Integration as proxy/middleware/logger, not inside the LLM.

* Reproducible and modular for testing or sandbox deployment.

Tasks (Phase 1):

1. Integration Paths:

Given access only to prompt, output, and meta-tags, the 5 most technically viable integration paths for a symbolic observer layer are:

* a) Network Proxy (Forward Proxy): SENTRY-LOGIC acts as an intermediary between the user/application and the LLM API. All requests and responses pass through it.

  * * Pros: Transparent to the application, centralized control.

  * * Cons: Potential latency, requires network configuration.

* b) Reverse Proxy: SENTRY-LOGIC sits in front of the LLM API endpoint, intercepting requests and responses.

  * * Pros: Server-side control, potentially easier to deploy with existing infrastructure.

  * * Cons: Requires control over the LLM API server.

* c) Middleware (API Gateway): If the LLM is accessed via an API gateway, SENTRY-LOGIC can be implemented as middleware within that gateway.

  * * Pros: Clean integration with existing API management, potentially good performance.

  * * Cons: Requires access to and modification of the API gateway.

* d) Logging Service (Sidecar/Agent): SENTRY-LOGIC runs as a separate service (e.g., a sidecar container) that receives logs from the application or LLM API. The application/API needs to be modified to send the necessary data.

  * * Pros: Decoupled from the main application/LLM, flexible deployment.

* * Cons: Requires application/API modification, potential for lost data if logging is not robust.

* e) Application-Level Integration (Library/SDK): SENTRY-LOGIC is provided as a library or SDK that the application developer integrates into their code. The application is responsible for sending the prompt, output, and meta-tags to SENTRY-LOGIC.

* * Pros: Highly flexible, allows for fine-grained control.

* * Cons: Requires application modification, potential for inconsistent integration.

2. Modular Structure:

A realistic modular structure for SENTRY-LOGIC is:

SENTRY-LOGIC

│

├── Input Handler

│ │

│ └── Receives prompt and meta-tags from integration path

│

├── Symbolic Analyzer

│ │

│ ├── Analyzes prompt, output, and meta-tags

│ │

│ └── Maps behavior to predefined symbols (Δ, Ω, Λ, ⇄)

│

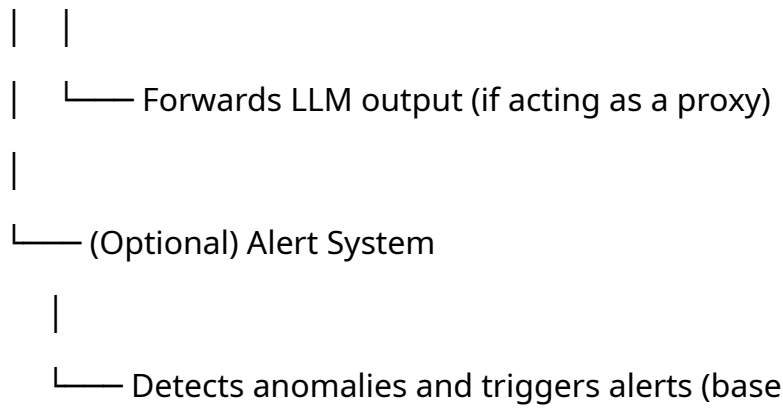
├── Log Generator

│ │

│ └── Creates structured logs with symbolic representations

│

├── Output Proxy



3. Data Extractable for Symbolic Mapping:

Realistically extractable data from prompt, output, and meta-tags, ranked in terms of feasibility:

* a) (High) Meta-tags: Role, policy, function call, data fetch (if provided by the LLM API or application). These are directly available and easy to access.

* b) (High) Prompt Structure: Identifying key phrases, intent, and entities within the prompt using regular expressions or simple parsing. This is relatively feasible with standard text processing techniques.

* c) (Medium) Output Structure: Analyzing the structure of the LLM's response (e.g., JSON, list, narrative) to infer the LLM's chosen output format. This is more complex but still achievable.

* d) (Medium) Semantic Similarity: Calculating the semantic similarity between the prompt and the output to detect potential rewrites or deviations in meaning (⇄). This requires more advanced techniques like sentence embeddings.

* e) (Low) Context Shifts (Δ): Inferring context shifts based on changes in topic, sentiment, or style between the prompt and the output. This is challenging and may require sophisticated natural language understanding (NLU).

* f) (Low) Policy Triggers (Ω): Detecting if the LLM's output triggered a specific policy (e.g., content moderation, safety guidelines) based on keywords or semantic analysis. This is very difficult without access to the LLM's internal policy mechanisms.

4. Failure Point and Mitigation:

The most likely failure point or blind spot is the accurate detection of context shifts (Δ) and policy triggers (Ω), as these require a deeper understanding of the LLM's internal state.

Mitigation strategies:

* Context Proxies: Use external knowledge bases or APIs to enrich the prompt and output with contextual information (e.g., topic classification, sentiment analysis). This can help infer context shifts.

* Role Estimation: If role meta-tags are unreliable or missing, attempt to estimate the LLM's assumed role based on the prompt and output (e.g., "helpful assistant," "code generator") using prompt/output analysis.

* Fine-grained Prompt Engineering: Encourage users to provide very detailed prompts, which can help in detecting context shifts.