Project NEMI: A Comprehensive Technical Overview and Implementation Guide
Document Version: 5.1
Date: October 13, 2025
Author: AI Agent

Revision History

- **Version 5.1 (October 13, 2025) - AI Agent**
  - **Enhanced Error Handling:** Overhauled the generateResponse method in GeminiClient.php to provide more robust and user-friendly error handling.
  - **Specific API Error Parsing:** The client now attempts to parse the JSON error response from the Gemini API to log specific error details (e.g., INVALID_ARGUMENT, PERMISSION_DENIED).
  - **User-Friendly Error Messages:** The system now returns a generic, user-friendly error message to the UI to avoid exposing raw technical details, while logging the full error for debugging purposes.
  - **Refined Exception Handling:** Improved the logic for handling different HTTP status codes and cURL failures, ensuring a clear distinction between internal application errors and external API failures.
- **Version 5.0 (October 12, 2025) - AI Agent**
  - Major Architectural Update: Transitioned from a purely keyword-based retrieval system to a Hybrid Search model, combining lexical (keyword) and semantic (vector) search for significantly improved context relevance.
  - Decoupled Embedding Logic: Introduced a new, separate EmbeddingClient.php to handle all vector embedding generation. This modular design allows for future swapping of embedding providers (e.g., to a local model) without altering core logic.
  - Overhauled Memory Retrieval: The getRelevantContext() method in MemoryManager.php was completely rewritten to perform the hybrid fusion of search results.
  - Semantic Memory Encoding: The updateMemory() method now automatically generates and stores a vector embedding for every new interaction.
  - Enhanced Configuration: Updated config.php with a master switch (ENABLE_EMBEDDINGS) to enable or disable the vector system, and tuning parameters (HYBRID_SEARCH_ALPHA, VECTOR_SEARCH_TOP_K) to control the search behavior.
  - UI Update: Version badge in index.php updated to v5.0.

1. Executive Summary

This document provides a comprehensive and multi-layered overview of "Project NEMI," an advanced AI initiative focused on developing a dynamic, persistent, and self-organizing memory system. The project's architecture has evolved significantly from a simple data logger into a sophisticated, reinforcement-based knowledge graph that leverages a Retrieval-Augmented Generation (RAG) framework.

This guide details the project's conceptual evolution, from its initial goals to its current state, and provides a complete technical specification for version 5.1. This version introduces a resilient Hybrid Search capability, fusing traditional keyword retrieval with modern semantic vector search. The system is designed to mimic key aspects of human memory, including the ability to prioritize relevant information, establish connections between concepts, and

gradually "forget" outdated or irrelevant data, thereby providing a robust foundation for a memory-driven AGI.

2. Architectural Evolution

The architecture of Project NEMI has progressed through several key versions, with each stage addressing the limitations of the previous one.

2.1. V0.0: Foundational Concept - The Simple Log

The project began with a straightforward objective: to create a persistent memory by logging all user and AI interactions in a structured JSON format.

- Goals:
    - Memory: Store all interactions.
    - Dynamic: Allow the memory to grow.
    - Efficient: Use a simple, machine-readable format.
    - JSON: The chosen format for data storage.
- Initial Structure: A single, flat JSON object where each interaction was keyed by a timestamp.
- Critique: This foundational approach, while functional, suffered from critical limitations in scalability, data retrieval (chronological-only), and a lack of relational context between entries.

2.2. V2.0: A Multi-Layered Memory Architecture

To overcome the limitations of V0.0, the architecture was redesigned to function more like a relational database, separating concerns into four distinct but interconnected components.

1. **Sessions:** Groups of interactions that provide episodic context to a conversation.
2. **Interactions:** The enriched core dialogue, containing not just raw text but processed input with identified intents, entities, sentiment, and contextual links.
3. **Entities (The Knowledge Graph):** A semantic memory store where the AI logs concepts, facts, and real-world entities separately from conversations. Each entity tracks every interaction where it was mentioned, creating an index for rapid, topic-based information retrieval.
4. **Users:** A personalization layer to store user-specific data and preferences.
   This structure transformed the memory from a linear log into a rich, interconnected graph of knowledge.

2.3. V3.0: Reinforcement-Based Memory and RAG

This version introduced the crucial concept of memory relevance, turning the static knowledge graph into a dynamic system that learns what information is important.

- Theory: An AI's memory should be continuously re-weighted based on interaction. A relevance score is attached to every piece of information, and this score dictates what is recalled and what is forgotten.
- Core Mechanisms:
    1. **Relevance Scoring:** Every interaction and entity is assigned a numerical score.
    2. **Reinforcement (Reward):** When a piece of memory is successfully used to provide context for a response, its relevance score is increased.

3. **Forgetting (Decay/Penalty):** A decay mechanism periodically reduces the score of memories that have not been accessed recently, causing them to become less likely to be recalled.
4. **Retrieval-Augmented Generation (RAG):** Before generating a response, the system queries its long-term memory for the highest-scoring information related to the current user input. This retrieved information is engineered into the final prompt, giving the AI a relevant "working memory."
5. **Pruning:** To manage storage size, a script removes the interactions with the lowest scores once a certain threshold is reached.

2.4. V4.0: Practical Implementation and the "Brain Analogy"

This version marked the critical transition from the theoretical V3.0 architecture to a tangible, functional application built in PHP. The core challenge was to translate the concepts of a dynamic knowledge graph, relevance scoring, and RAG into a resilient and modular software structure.

- Theory: The entire memory and response workflow can be modeled after a biological brain with distinct components for conscious thought, memory management, and communication. This architectural pattern, termed the "Brain Analogy," promotes a clear separation of concerns and makes the complex system easier to manage and extend.
- Core Mechanisms:
    1. **Orchestration (index.php):** Acts as the "conscious thought," handling the immediate user request and coordinating the entire sequence of operations: recalling memories, building the prompt, getting the AI response, and updating the memory.
    2. **Memory System (MemoryManager.php):** Represents the core brain, responsible for all memory logic. It implements the keyword-based retrieval, the reward/decay scoring system from V3.0, and the pruning of old memories.
    3. **Communication (GeminiClient.php):** Functions as the "senses and voice," managing all outbound API calls to the generative AI model and parsing its responses.
    4. **Rules & Personality (config.php):** A centralized configuration file that acts as the brain's "personality," defining all the rules for learning, forgetting, and responding.
- Key Features: This version successfully implemented the complete reinforcement learning loop in code, including the relevance scoring, decay mechanism, user feedback loop, and dynamic tool selection based on user input. Retrieval at this stage was purely lexical (keyword and entity-based), setting the foundation for the more advanced semantic capabilities introduced in later versions.

2.5. V5.0: Hybrid Search and Decoupled Architecture

This version represents a major leap in retrieval intelligence, moving beyond the limitations of purely lexical search to a more human-like conceptual understanding.

- Theory: The most relevant memories are found by combining the precision of keyword search with the conceptual breadth of semantic search. A memory's relevance is no longer just its score but also its contextual similarity to the user's query.
- Core Mechanisms:

1. **Vector Embeddings:** Every interaction, upon creation, is converted into a numerical vector that represents its semantic meaning. This vector is stored alongside the interaction data.
2. **Hybrid Retrieval:** The context retrieval process is now a three-stage pipeline:
    - **Vector Search:** The user's query is converted into a vector. The system calculates the "cosine similarity" between this query vector and all stored memory vectors to find the most conceptually similar memories.
    - **Keyword Search:** The original keyword and entity search is performed to find memories with exact lexical matches and high relevance scores.
    - **Fusion:** The results from both searches are merged and re-ranked using a weighted algorithm (HYBRID_SEARCH_ALPHA) to produce a final list that is both semantically relevant and lexically precise.
3. **Decoupled Embedding Client:** To ensure resilience and future-proofing, all embedding logic is isolated into a dedicated EmbeddingClient.php. This allows the embedding provider (e.g., Gemini API, a local model) to be changed with minimal impact on the rest of the application.

3. Technical Specification and Implementation: Version 5.1
Version 5.1 is a mature PHP implementation of the Hybrid Search RAG architecture.

3.1. The "Brain Analogy": System Components
The project is structured into modular components, each with a distinct role:

- **index.php (Conscious Thought):** The main orchestrator. It handles the user request, coordinates with the memory and AI client, builds the final prompt, and renders the UI.
- **MemoryManager.php (The Memory System):** The core brain. It manages all memory operations, including loading, saving, and the complex hybrid retrieval of relevant context.
- **GeminiClient.php (Senses & Voice):** The dedicated communicator for generating content. It handles all conversational interactions with the Google Gemini API.
- **EmbeddingClient.php (The Conceptual Librarian):** A new, dedicated communicator for generating embeddings. It translates text into its semantic meaning (vectors) and is decoupled from the GeminiClient.
- **config.php (Personality & Rules):** A centralized configuration file defining API keys, file paths, and the constants that govern the memory and hybrid search logic.

3.2. Key Features and Enhancements in V5.1

- **Hybrid Search System:** The core memory retrieval engine now combines keyword-based lexical search with vector-based semantic search, providing superior contextual recall.
- **Decoupled Embedding Logic:** A new EmbeddingClient.php isolates all embedding-related API calls, making the system resilient and easy to adapt to other embedding providers (e.g., local models via Ollama).
- **Configurable Search Behavior:** The config.php file contains a master switch (ENABLE_EMBEDDINGS) to turn the entire vector system on or off, as well as

tuning parameters (HYBRID_SEARCH_ALPHA) to balance between keyword and semantic results.

- **Automatic Semantic Encoding:** The updateMemory() process now automatically creates a vector embedding for each new conversation turn, ensuring all new memories are semantically searchable.
- **Robust Error Handling:** The GeminiClient.php now features enhanced error handling, providing clearer, user-friendly messages to the UI while logging detailed technical information for easier debugging.

4. Complete Source Code (Version 5.1)

config.php

code PHP
downloadcontent_copy
expand_less

```php
<?php

// --- API Configuration ---
define('GEMINI_API_KEY', 'YOUR_GEMINI_API_KEY_HERE'); // <-- REPLACE THIS
define('MODEL_ID', 'gemini-1.5-flash-latest');
define('API_ENDPOINT', 'generateContent');

// --- NEW: Embedding Configuration ---
define('ENABLE_EMBEDDINGS', true); // Master switch to enable/disable
vector search.
define('EMBEDDING_MODEL_ID', 'text-embedding-004'); // Specialized model
for embeddings.

// --- File Paths ---
define('DATA_DIR', __DIR__ . '/data');
define('INTERACTIONS_FILE', DATA_DIR . '/interactions.json');
define('ENTITIES_FILE', DATA_DIR . '/entities.json');
define('PROMPTS_LOG_FILE', DATA_DIR . '/prompts.json');

// --- Memory Logic Configuration ---
define('REWARD_SCORE', 0.5);
define('DECAY_SCORE', 0.05);
define('INITIAL_SCORE', 1.0);
define('PRUNING_THRESHOLD', 500);
define('CONTEXT_TOKEN_BUDGET', 4000);

// --- NEW: Hybrid Search Tuning ---
// Balances between keyword (relevance_score) and semantic (similarity)
search.
// 0.0 = Pure keyword search. 1.0 = Pure semantic search. 0.5 is balanced.
define('HYBRID_SEARCH_ALPHA', 0.5);
// The number of top semantic results to fetch before ranking.
define('VECTOR_SEARCH_TOP_K', 15);


// --- Advanced Scoring & Relationships ---
define('NOVELTY_BONUS', 0.3);
define('RELATIONSHIP_STRENGTH_INCREMENT', 0.1);
define('RECENT_TOPIC_DECAY_MODIFIER', 0.1);
define('USER_FEEDBACK_REWARD', 0.5);
define('USER_FEEDBACK_PENALTY', -0.5);
```

EmbeddingClient.php (New File)

code PHP
downloadcontent_copy
expand_less

```
    <?php

/**
 * Handles the generation of vector embeddings via the Gemini API.
 * This class is decoupled from the main content generation client
 * to allow for easy swapping with other embedding providers (e.g., local
Ollama).
 */
class EmbeddingClient
{
    private string $apiKey;
    private string $modelId;
    private string $apiUrl;

    public function __construct()
    {
        $this->apiKey = GEMINI_API_KEY;
        $this->modelId = EMBEDDING_MODEL_ID;
        $this->apiUrl =
"https://generativelanguage.googleapis.com/v1beta/models/{$this-
>modelId}:embedContent?key={$this->apiKey}";
    }

    /**
     * Converts a string of text into a vector embedding.
     *
     * @param string $text The text to embed.
     * @return array|null The vector as an array of floats, or null on
error.
     */
    public function getEmbedding(string $text): ?array
    {
        if (!ENABLE_EMBEDDINGS) {
            return null; // Return null if embeddings are disabled in
config
        }

        try {
            $requestBody = [
                'model'   => "models/{$this->modelId}",
                'content' => ['parts' => [['text' => $text]]]
            ];

            $ch = curl_init();
            curl_setopt($ch, CURLOPT_URL, $this->apiUrl);
            curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
            curl_setopt($ch, CURLOPT_POST, true);
            curl_setopt($ch, CURLOPT_POSTFIELDS,
json_encode($requestBody));
            curl_setopt($ch, CURLOPT_HTTPHEADER, ['Content-Type:
application/json']);

            $rawResponse = curl_exec($ch);
            if (curl_errno($ch)) {
```

```
                throw new Exception("cURL Error getting embedding: " .
curl_error($ch));
            }
            curl_close($ch);

            $decodedResponse = json_decode($rawResponse, true);
            $embedding = $decodedResponse['embedding']['values'] ?? null;

            if ($embedding === null) {
                // Log the actual response for debugging
                error_log("Failed to find embedding in API response: " .
$rawResponse);
                throw new Exception("Could not find embedding in API
response.");
            }

            return $embedding;
        } catch (Exception $e) {
            error_log($e->getMessage());
            return null;
        }
    }
}
```

GeminiClient.php

code PHP
downloadcontent_copy
expand_less

```
    <?php

class GeminiClient
{
    private string $apiKey;
    private string $modelId;
    private string $apiUrl;

    public function __construct()
    {
        $this->apiKey = GEMINI_API_KEY;
        $this->modelId = MODEL_ID;
        $this->apiUrl =
"https://generativelanguage.googleapis.com/v1beta/models/{$this->modelId}:"
. API_ENDPOINT . "?key={$this->apiKey}";
    }

    private function logPrompt(string $prompt, array $requestBody, string
$rawResponse, ?string $error = null): void
    {
        if (!file_exists(PROMPTS_LOG_FILE)) {
            file_put_contents(PROMPTS_LOG_FILE, '[]');
        }
        $logData = json_decode(file_get_contents(PROMPTS_LOG_FILE), true);
        $decodedResponse = json_decode($rawResponse, true);

        $logEntry = [
            'timestamp' => date('c'),
            'request' => [
                'model' => $this->modelId,
```

```
                'tools' => array_keys(array_column($requestBody['tools'] ??
[], null, 0)),
                'prompt_text' => $prompt
            ],
            'response' => [
                'token_usage' => $decodedResponse['usageMetadata'] ?? null,
                'finish_reason' =>
$decodedResponse['candidates'][0]['finishReason'] ?? null,
                'response_text' =>
$decodedResponse['candidates'][0]['content']['parts'][0]['text'] ?? null,
            ],
            'error' => $error
        ];

        $logData[] = $logEntry;
        file_put_contents(PROMPTS_LOG_FILE, json_encode($logData,
JSON_PRETTY_PRINT | JSON_UNESCAPED_SLASHES));
    }

    public function generateResponse(string $prompt, array $enabledTools =
[]): string
    {
        if (empty($this->apiKey) || $this->apiKey ===
'YOUR_GEMINI_API_KEY_HERE') {
            return "ERROR: Gemini API Key is not configured in config.php.
Please set a valid key to proceed.";
        }

        $rawResponse = '';
        $requestBody = [];

        try {
            $requestBody = [
                'contents' => [['role' => 'user', 'parts' => [['text' =>
$prompt]]]],
                'generationConfig' => ['maxOutputTokens' => 8192],
            ];

            $requestTools = [];
            if (in_array('googleSearch', $enabledTools)) {
                $requestTools[] = ['googleSearch' => new stdClass()];
            }

            if (!empty($requestTools)) {
                 $requestBody['tools'] = $requestTools;
            }

            $ch = curl_init();
            curl_setopt($ch, CURLOPT_URL, $this->apiUrl);
            curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
            curl_setopt($ch, CURLOPT_POST, true);
            curl_setopt($ch, CURLOPT_POSTFIELDS,
json_encode($requestBody));
            curl_setopt($ch, CURLOPT_HTTPHEADER, ['Content-Type:
application/json']);

            $rawResponse = curl_exec($ch);

            if (curl_errno($ch)) { throw new Exception("cURL Error: " .
curl_error($ch)); }
```

```
            $httpCode = curl_getinfo($ch, CURLINFO_HTTP_CODE);
            if ($httpCode !== 200) { throw new Exception("API Error: HTTP
{$httpCode}. Response: " . $rawResponse); }
            curl_close($ch);

            $decodedResponse = json_decode($rawResponse, true);
            if (json_last_error() !== JSON_ERROR_NONE) { throw new
Exception("JSON Decode Error for API response."); }

            $responseText =
$decodedResponse['candidates'][0]['content']['parts'][0]['text'] ?? null;
            if ($responseText === null) {
                throw new Exception("Could not find text part in the API
response. The response may contain a tool call or be empty.");
            }

            $this->logPrompt($prompt, $requestBody, $rawResponse);
            return $responseText;

        } catch (Exception $e) {
            $detailedError = $e->getMessage();
            $userFriendlyError = "An unexpected error occurred while
communicating with the AI. Please check the system logs for more details.";

            // Attempt to parse a more specific error message from the
API's raw response
            if (!empty($rawResponse)) {
                $decodedError = json_decode($rawResponse, true);
                if (isset($decodedError['error']['message'])) {
                    $apiErrorMsg = $decodedError['error']['message'];
                    $detailedError = "API Error: " . $apiErrorMsg; //
Overwrite for cleaner logs

                    // Tailor the user-facing message for common,
understandable errors
                    if (isset($decodedError['error']['status'])) {
                        switch ($decodedError['error']['status']) {
                            case 'INVALID_ARGUMENT':
                                $userFriendlyError = "There was an issue
with the request sent to the AI, possibly due to a configuration problem or
invalid input.";
                                break;
                            case 'PERMISSION_DENIED':
                                $userFriendlyError = "Authentication with
the AI service failed. Please verify that the API key is correct and has
the necessary permissions.";
                                break;
                        }
                    }
                }
            }

            // Log the detailed, technical error and return the safe, user-
friendly one.
            $this->logPrompt($prompt, $requestBody, $rawResponse,
$detailedError);
            return "ERROR: " . $userFriendlyError;
        }
    }
}
```

index.php

code PHP
downloadcontent_copy
expand_less

```
    <?php
ini_set('display_errors', 1);
error_reporting(E_ALL);

require_once 'config.php';
require_once 'MemoryManager.php';
require_once 'GeminiClient.php';

// Handle User Feedback
if ($_SERVER["REQUEST_METHOD"] == "GET" && isset($_GET['feedback'],
$_GET['id'])) {
    $memory = new MemoryManager();
    $isGood = $_GET['feedback'] === 'good';
    $memory->applyFeedback($_GET['id'], $isGood);
    $memory->saveMemory();
    header("Location: index.php");
    exit();
}

function runCoreLogic(string $userInput): array
{
    $memory = new MemoryManager();
    $gemini = new GeminiClient();

    // Dynamic Tool Selection Logic
    $toolsToUse = ['googleSearch']; // Enable search by default
    $urlPattern = '/\b(https?|ftp|file):\/\/[-A-Z0-9+&@#\/%?=~_|!:,.;]*[-A-
Z0-9+&@#\/%=~_|]/i';
    if (preg_match($urlPattern, $userInput)) {
        // The prompt guides the AI to use its search tool to analyze the
URL
        $userInput .= "\n\n[SYSTEM NOTE: The user has provided a URL.
Prioritize analyzing its content.]";
    }

    $recalled = $memory->getRelevantContext($userInput);
    $context = $recalled['context'];

    $systemPrompt = $memory->getTimeAwareSystemPrompt();
    $currentTime = "CURRENT_TIME: " . date('Y-m-d H:i:s T');
    $finalPrompt = "{$systemPrompt}\n\n---RECALLED CONTEXT---\n{$context}--
-END CONTEXT---\n\n{$currentTime}\n\nUser query: \"{$userInput}\"";

    $aiResponse = $gemini->generateResponse($finalPrompt, $toolsToUse);

    $newInteractionId = $memory->updateMemory($userInput, $aiResponse,
$recalled['used_interaction_ids']);
    $memory->saveMemory();

    return [
        'userInput' => $userInput,
        'aiResponse' => $aiResponse,
        'context' => $context,
        'interactionId' => $newInteractionId
    ];
```

```
}

// --- Main Web Mode Execution ---
$userInput = '';
$aiResponse = '';
$context = '';
$interactionId = '';

if ($_SERVER["REQUEST_METHOD"] == "POST" && !empty($_POST['prompt'])) {
    $userInput = trim($_POST['prompt']);
    $result = runCoreLogic($userInput);
    $aiResponse = $result['aiResponse'];
    $context = $result['context'];
    $interactionId = $result['interactionId'];
}
?>
<!DOCTYPE html>
<html lang="en" data-bs-theme="dark">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Project NEMI v5.1 (Hybrid Search)</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.c
ss" rel="stylesheet">
    <style>
        body { background-color: #212529; color: #e9ecef; }
        .container { max-width: 800px; }
        .card { border-color: #495057; }
        .card-header { background-color: #343a40; }
        .feedback-buttons a { margin-right: 10px; }
    </style>
</head>
<body>
    <div class="container mt-5">
        <div class="text-center mb-4">
            <h1 class="display-4">Project NEMI <span class="badge bg-
success">v5.1</span></h1>
            <p class="lead text-muted">AI with Hybrid Vector Search</p>
        </div>
        <div class="card bg-dark mb-3">
            <div class="card-body">
                <form action="index.php" method="post">
                    <textarea class="form-control bg-dark text-white mb-3"
name="prompt" rows="3" placeholder="Ask anything or provide a URL to
analyze..."><?= htmlspecialchars($userInput) ?></textarea>
                    <button type="submit" class="btn btn-primary w-
100">Send</button>
                </form>
            </div>
        </div>
        <?php if ($_SERVER["REQUEST_METHOD"] == "POST"): ?>
            <div class="card bg-dark mb-3">
                <div class="card-header">Your Prompt</div>
                <div class="card-body"><p class="card-text"><?=
nl2br(htmlspecialchars($userInput)) ?></p></div>
            </div>
            <div class="card bg-dark mb-3">
                <div class="card-header d-flex justify-content-between
align-items-center">
                    AI Response
```

```
                    <div class="feedback-buttons">
                        <a href="?feedback=good&id=<?=
urlencode($interactionId) ?>" class="btn btn-sm btn-outline-success">👍
Good</a>
                        <a href="?feedback=bad&id=<?=
urlencode($interactionId) ?>" class="btn btn-sm btn-outline-danger">👎
Bad</a>
                    </div>
                </div>
                <div class="card-body"><p class="card-text"><?=
nl2br(htmlspecialchars($aiResponse)) ?></p></div>
            </div>
            <div class="card bg-dark">
                <div class="card-header"><a class="text-decoration-none"
data-bs-toggle="collapse" href="#debugCollapse">Debug: Recalled
Context</a></div>
                <div class="collapse" id="debugCollapse">
                    <div class="card-body"><pre class="text-white-50
small"><code><?= htmlspecialchars($context) ?></code></pre></div>
                </div>
            </div>
        <?php endif; ?>
    </div>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.
min.js"></script>
</body>
</html>
```

MemoryManager.php

code PHP
downloadcontent_copy
expand_less

```
    <?php
// Now requires the new EmbeddingClient
require_once 'EmbeddingClient.php';

class MemoryManager
{
    private array $interactions = [];
    private array $entities = [];
    private ?EmbeddingClient $embeddingClient;

    public function __construct()
    {
        if (!is_dir(DATA_DIR)) {
            mkdir(DATA_DIR, 0775, true);
        }
        $this->loadMemory();

        // Instantiate the client only if embeddings are enabled.
        // This is where you would swap in a different client (e.g., new
OllamaEmbeddingClient()).
        $this->embeddingClient = ENABLE_EMBEDDINGS ? new EmbeddingClient()
: null;
    }
```

```
    private function loadMemory(): void
    {
        $this->interactions = file_exists(INTERACTIONS_FILE) ?
json_decode(file_get_contents(INTERACTIONS_FILE), true) : [];
        $this->entities = file_exists(ENTITIES_FILE) ?
json_decode(file_get_contents(ENTITIES_FILE), true) : [];
    }

    public function saveMemory(): void
    {
        ksort($this->interactions);
        ksort($this->entities);
        $options = JSON_PRETTY_PRINT | JSON_UNESCAPED_SLASHES;
        file_put_contents(INTERACTIONS_FILE, json_encode($this-
>interactions, $options));
        file_put_contents(ENTITIES_FILE, json_encode($this->entities,
$options));
    }

    // --- Core Memory Retrieval (HYBRID SEARCH) ---

    /**
     * Calculates the cosine similarity between two vectors.
     * @return float A value between -1 and 1. Higher is more similar.
     */
    private function cosineSimilarity(array $vecA, array $vecB): float
    {
        $dotProduct = 0.0;
        $magA = 0.0;
        $magB = 0.0;
        $count = count($vecA);

        for ($i = 0; $i < $count; $i++) {
            $dotProduct += $vecA[$i] * $vecB[$i];
            $magA += $vecA[$i] * $vecA[$i];
            $magB += $vecB[$i] * $vecB[$i];
        }

        $magA = sqrt($magA);
        $magB = sqrt($magB);

        if ($magA == 0 || $magB == 0) {
            return 0;
        }

        return $dotProduct / ($magA * $magB);
    }

    public function getRelevantContext(string $userInput): array
    {
        // --- Vector Search (Semantic) ---
        $semanticResults = [];
        if ($this->embeddingClient !== null) {
            $inputVector = $this->embeddingClient-
>getEmbedding($userInput);
            if ($inputVector !== null) {
                $similarities = [];
                foreach ($this->interactions as $id => $interaction) {
                    if (isset($interaction['embedding']) &&
is_array($interaction['embedding'])) {
```

```
                        $similarity = $this->cosineSimilarity($inputVector,
$interaction['embedding']);
                        $similarities[$id] = $similarity;
                    }
                }
                // Sort by similarity score, descending
                arsort($similarities);
                $semanticResults = array_slice($similarities, 0,
VECTOR_SEARCH_TOP_K, true);
            }
        }

        // --- Keyword Search (Lexical) ---
        $inputEntities = $this->extractEntities($userInput);
        $searchEntities = $this-
>normalizeAndExpandEntities($inputEntities);
        $keywordResults = [];
        foreach ($searchEntities as $entityKey) {
            if (isset($this->entities[$entityKey])) {
                foreach ($this->entities[$entityKey]['mentioned_in'] as
$id) {
                    // Store the relevance score for ranking
                    if(isset($this->interactions[$id])) {
                         $keywordResults[$id] = $this-
>interactions[$id]['relevance_score'];
                    }
                }
            }
        }
        arsort($keywordResults);

        // --- Hybrid Fusion ---
        $fusedScores = [];
        $allIds = array_unique(array_merge(array_keys($semanticResults),
array_keys($keywordResults)));

        foreach ($allIds as $id) {
            $semanticScore = $semanticResults[$id] ?? 0.0;
            // Normalize relevance score to be roughly between 0 and 1 for
better blending
            $relevanceScore = isset($keywordResults[$id]) ?
tanh($keywordResults[$id] / 10) : 0.0;

            // Weighted average of both scores
            $fusedScores[$id] = (HYBRID_SEARCH_ALPHA * $semanticScore) +
((1 - HYBRID_SEARCH_ALPHA) * $relevanceScore);
        }
        arsort($fusedScores);

        // --- Build Context from Fused Results ---
        $context = '';
        $tokenCount = 0;
        $usedInteractionIds = [];
        foreach ($fusedScores as $id => $score) {
            if (!isset($this->interactions[$id])) continue;
            $memory = $this->interactions[$id];
            $timestamp = date('Y-m-d H:i:s',
strtotime($memory['timestamp']));
            $memoryText = "[On {$timestamp}] User:
'{$memory['user_input_raw']}'. You: '{$memory['ai_output']}'.\n";
            $memoryTokenCount = str_word_count($memoryText);
```

```
            if ($tokenCount + $memoryTokenCount <= CONTEXT_TOKEN_BUDGET) {
                $context .= $memoryText;
                $tokenCount += $memoryTokenCount;
                $usedInteractionIds[] = $id;
            } else {
                break;
            }
        }

        return [
            'context' => empty($context) ? "No relevant memories found.\n"
: $context,
            'used_interaction_ids' => $usedInteractionIds
        ];
    }

    public function updateMemory(string $userInput, string $aiOutput, array
$usedInteractionIds): string
    {
        $recentEntities = [];
        foreach ($usedInteractionIds as $id) {
            if (isset($this->interactions[$id])) {
                $this->interactions[$id]['relevance_score'] +=
REWARD_SCORE;
                $this->interactions[$id]['last_accessed'] = date('c');
                if (isset($this-
>interactions[$id]['processed_input']['keywords'])) {
                    $recentEntities = array_merge($recentEntities, $this-
>interactions[$id]['processed_input']['keywords']);
                }
            }
        }
        $recentEntities = array_unique($recentEntities);

        foreach ($this->interactions as &$interaction) {
            $keywords = $interaction['processed_input']['keywords'] ?? [];
            $isRelatedToRecentTopic = !empty(array_intersect($keywords,
$recentEntities));
            $decay = $isRelatedToRecentTopic ? DECAY_SCORE *
RECENT_TOPIC_DECAY_MODIFIER : DECAY_SCORE;
            $interaction['relevance_score'] -= $decay;
        }

        $newId = uniqid('int_', true);
        $keywords = $this->extractEntities($userInput);

        // --- NEW: Generate and store embedding for the new interaction --
-
        $embedding = null;
        if ($this->embeddingClient !== null) {
            $fullText = "User: {$userInput} | AI: {$aiOutput}";
            $embedding = $this->embeddingClient->getEmbedding($fullText);
        }

        $this->interactions[$newId] = [
            'timestamp' => date('c'),
            'user_input_raw' => $userInput,
            'processed_input' => ['keywords' => $keywords],
            'ai_output' => $aiOutput,
            'relevance_score' => INITIAL_SCORE,
```

```
            'last_accessed' => date('c'),
            'context_used_ids' => $usedInteractionIds,
            'embedding' => $embedding // Add the new embedding to the
record
        ];

        $this->updateEntitiesFromInteraction($keywords, $newId);
        $this->pruneMemory();

        return $newId;
    }

    private function updateEntitiesFromInteraction(array $keywords, string
$interactionId): void
    {
        $isNovel = false;
        foreach ($keywords as $keyword) {
            $entityKey = strtolower($keyword);
            if (!isset($this->entities[$entityKey])) {
                $isNovel = true;
                $this->entities[$entityKey] = [
                    'name' => $keyword, 'type' => 'Concept', 'access_count'
=> 0, 'relevance_score' => INITIAL_SCORE,
                    'mentioned_in' => [], 'relationships' => []
                ];
            }
            $this->entities[$entityKey]['access_count']++;
            $this->entities[$entityKey]['relevance_score'] += REWARD_SCORE;
            if (!in_array($interactionId, $this-
>entities[$entityKey]['mentioned_in'])) {
                $this->entities[$entityKey]['mentioned_in'][] =
$interactionId;
            }
        }

        if ($isNovel) {
            $this->interactions[$interactionId]['relevance_score'] +=
NOVELTY_BONUS;
        }

        if (count($keywords) > 1) {
            foreach ($keywords as $k1) {
                foreach ($keywords as $k2) {
                    if ($k1 !== $k2) {
                        $this->entities[$k1]['relationships'][$k2] =
($this->entities[$k1]['relationships'][$k2] ?? 0) +
RELATIONSHIP_STRENGTH_INCREMENT;
                    }
                }
            }
        }
    }

    public function applyFeedback(string $interactionId, bool $isGood):
void
    {
        if (!isset($this->interactions[$interactionId])) return;

        $adjustment = $isGood ? USER_FEEDBACK_REWARD :
USER_FEEDBACK_PENALTY;
```

```
        $this->interactions[$interactionId]['relevance_score'] +=
$adjustment;

        $contextIds = $this-
>interactions[$interactionId]['context_used_ids'] ?? [];
        foreach ($contextIds as $id) {
            if (isset($this->interactions[$id])) {
                $this->interactions[$id]['relevance_score'] += $adjustment
/ 2;
            }
        }
    }

    private function pruneMemory(): void
    {
        if (count($this->interactions) > PRUNING_THRESHOLD) {
            uasort($this->interactions, fn($a, $b) =>
($a['relevance_score'] ?? 1.0) <=> ($b['relevance_score'] ?? 1.0));
            $this->interactions = array_slice($this->interactions,
count($this->interactions) - PRUNING_THRESHOLD, null, true);
        }
    }

    public function getTimeAwareSystemPrompt(): string
    {
        return "**PRIMARY DIRECTIVE: YOU ARE A HELPFUL, TIME-AWARE
ASSISTANT.**\n\n" .
            "**RULES OF OPERATION:**\n" .
            "1.  **ANALYZE TIMESTAMPS:** You will be given a `CURRENT_TIME`
and `RECALLED_CONTEXT`. Use this to understand the history of events.\n" .
            "2.  **CALCULATE RELATIVE TIME:** Interpret expressions like
'yesterday' against the provided `CURRENT_TIME`.\n\n" .
            "**TOOL EXECUTION MANDATE:**\n" .
            "3.  **DIRECTLY USE TOOLS:** You have a `googleSearch` tool.
Your primary goal is to use this tool to directly answer the user's
question. **DO NOT describe that you are going to use a tool.** Execute it
and provide the final answer based on its output.\n" .
            "4.  **FULFILL THE REQUEST:** If the user provides a URL, use
your search ability to access its content and provide a summary. If they
ask a general question, use search to find the answer.";
    }

    private function extractEntities(string $text): array
    {
        $text = strtolower($text);
        $text = preg_replace('/https?:\/\/[^\s]+/', ' ', $text);
        $words = preg_split('/[\s,\.\?\!\[\]:]+/', $text);
        $stopWords = ['a', 'an', 'the', 'is', 'in', 'it', 'of', 'for',
'on', 'what', 'were', 'my', 'that', 'we', 'to', 'user', 'note', 'system',
'please'];
        return array_filter(array_unique($words), fn($word) =>
!in_array($word, $stopWords) && strlen($word) > 3);
    }

    private function normalizeAndExpandEntities(array $baseEntities): array
    {
        $expanded = $baseEntities;
        foreach ($baseEntities as $entityKey) {
            if (isset($this->entities[$entityKey]['relationships'])) {
                $expanded = array_merge($expanded, array_keys($this-
>entities[$entityKey]['relationships']));
```

```
            }
        }
        return array_unique($expanded);
    }
}
```

Nota Bene: Rules for Future Document Updates
AI and/or agents must adhere to the following rules to maintain the consistency and integrity of this document.

1. **Strict Versioning:** All modifications, regardless of size, mandate a version increment.
    - **Patch (e.g., 4.2 -> 4.2.1):** Typographical corrections, code comments, non-functional changes.
    - **Minor (e.g., 4.2 -> 4.3):** New features, changes in logic, enhancements that are backward-compatible.
    - **Major (e.g., 4.0 -> 5.0):** Significant architectural changes or rewrites that are not backward-compatible.
2. **Mandatory Changelog:** Every new version must be documented in a "Revision History" section added to the top of the document. The entry must include the new version number, date, author, and a concise, bulleted list of all changes made.
3. **Code Synchronization:** The source code presented within this document must be an exact and complete representation of the functional source code files. Any change to a .php file must be immediately and accurately reflected in its corresponding code block herein.
4. **Preserve Architecture:** New functionality must be integrated into the existing "Brain Analogy" component structure (index.php, MemoryManager.php, GeminiClient.php, config.php). If a new component is necessary, its purpose and interaction with the existing components must be explicitly defined.
5. **Centralize Configuration:** All new constants, thresholds, or configurable parameters must be added exclusively to config.php. No hardcoded "magic values" are permitted in the logic of other files.
6. **Maintain Structural Integrity:** The existing numbered heading structure of this document must be preserved. New content should be added as subsections within the relevant existing section. Do not alter the primary document flow.
7. **Direct and Specific Language:** All descriptions of changes must be technical, direct, and unambiguous. Clearly state what was changed, why it was changed, and reference the specific function or component affected.