iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** I **Month of publication:** January 2026

DOI: https://doi.org/10.22214/ijraset.2026.78270

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue I Jan 2026- Available at www.ijraset.com

ForgetAgent: Verifiable Deletion in Multi-Layer Memory Architectures for LLM Agents

Prerna Singhal
Empros International School, Pune, Maharashtra, India

Abstract: ***Autonomous agents built on large language models increasingly rely on persistent memory to support long-horizon reasoning, personalization, and tool-augmented decision-making. However, current agent architectures generally lack strong guarantees that deleted memories are unrecoverable under adversarial querying, paraphrastic retrieval, or indirect inference [3,4,16]. We introduce ForgetAgent, a formal framework and reference implementation for verifiable deletion in heterogeneous agent memory systems. We first develop a system model in which agent memory spans embeddings, structured records, caches, derived entities, and tool transcripts, and define a threat model encompassing adversarial users, compromised agents, and external attackers capable of semantic, paraphrastic, neighborhood-based, and narrative (story-completion) retrieval. Building on this model, we formalize (ε, δ) -style deletion correctness, utility preservation, and computational efficiency, and propose a seven-layer deletion pipeline that combines dependency-graph–based identification, raw storage purging, tombstone embeddings with neighborhood reweighting, cache and derived-entity invalidation, and transcript sanitization, together with cryptographic deletion receipts and automated red-team verification. Our red-team library instantiates membership inference, paraphrase retrieval, neighbor leakage, transcript analysis, and story-completion attacks, providing a comprehensive black-box evaluation of residual information flow. To enable reproducible assessment, we release ForgetAgentBench, a benchmark of 500 synthetic agent interactions with 1,500 labeled deletion targets spanning personal, preference, and domain-specific memories at multiple difficulty levels. In experiments on ForgetAgentBench, naive deletion achieves 18% robustness to attack, while our full layered method reaches 94% robustness with 97% retained task utility; however, these results are limited to synthetic benchmark settings. These results demonstrate that verifiable, multi-layer deletion is both necessary and feasible for trustworthy, privacy- and regulation-compliant LLM agents and establish a concrete foundation for future work on principled memory control in agentic systems.***

Keywords: ***Machine unlearning, agent memory, right-to-be-forgotten, membership inference, trustworthy AI, retrieval-augmented generation, privacy-preserving systems***

I. INTRODUCTION

A. Problem Statement

Autonomous agents augmented with LLMs have become increasingly practical in enterprise and consumer settings [3,14]. Unlike stateless chatbots, these agents maintain persistent memory of past interactions, user preferences, system state, and tool outputs [13], [14]. This memory is essential: it enables the agent to reason about long-horizon tasks, avoid repeating errors, and provide personalized behavior [13], [14]. However, this same capability introduces a critical vulnerability.

When a user requests deletion of their data—whether driven by privacy law (GDPR, CCPA), contractual obligations (RTBF requests), or safety concerns [1], [2], [5] (removal of harmful information)—current agent systems lack a formal mechanism to guarantee that the deleted information cannot be recovered.

A user's deleted personal data may persist in:

- Raw text storage (e.g., in a document database or log)
- Vector embeddings in semantic search indices
- Summarized or compressed representations
- Derived entities (e.g., inferred user preferences, extracted facts)
- Cached tool outputs or API responses
- Temporary reasoning state in the agent's runtime context (e.g., scratchpads or reasoning traces)

Even when one storage layer is cleared, the deleted information may leak through alternative retrieval paths [3], [4], [16]. For instance, an adversary may craft paraphrased queries that semantically match the deleted memory, reconstruct information from embedded vector space using membership inference attacks[4], or exploit tool call chains that reference deleted context.

The core research question we address is: *How can we verify that deleted agent memory is unrecoverable under adversarial prompting, semantic retrieval, and indirect multi-hop inference attacks*?

B. Why Agent Memory Deletion is Different from Model Unlearning

Recent machine unlearning research[1,2]—particularly on LLMs—focuses on removing training data influence from model weights (or from RAG indices in retrieval-augmented systems).

This work assumes:

- A well-defined "forget set" of training data
- Access to a retrained oracle model as ground truth
- Evaluation via membership inference attacks on model outputs[4]
- One-time or batch unlearning operations

Agent memory deletion operates under fundamentally different constraints:

1) Dynamic memory topology: Agent memory is not a static training set but a continuously evolving graph of episodic, semantic, and procedural knowledge. A single "memory item" (e.g., "User prefers vegetarian meals") may be instantiated across multiple representations and reasoning chains.

2) Heterogeneous storage: Unlike unlearning in models or unified RAG systems, agent memory spans vector databases, structured SQL records, time-series logs, derived cached summaries, and tool-call receipts [9], [11], [14]. No single "delete" operation reaches all layers.

3) Semantic reconstruction risk: Agent reasoning can reconstruct deleted information indirectly. For example, deleting "User visited Hospital X on date Y" does not prevent the agent from inferring it through tool call logs showing "Map API queried for directions to Hospital X on date Y" combined with calendar entries.

4) Verifiability under multi-agent inference: In multi-agent systems, deleted information from one agent's memory may be reconstructed through cross-agent communication. This requires formal verification of information flow properties across agent boundaries [13], [14].

5) No accessible ground truth: Unlike model unlearning where we can retrain from scratch,[2] agent memory systems are often proprietary, deployed on customers' infrastructure, or too expensive to fully retrain. Evaluation must rely on black-box attacks and indirect signals.

This paper bridges this gap by proposing ForgetAgent, a framework that:

- Formally defines threat models for agent memory deletion
- Identifies attack surface across all memory layers
- Proposes evaluation metrics grounded in practical adversarial scenarios
- Releases open benchmarks and baseline implementations
- Provides practitioners with auditable deletion protocols

C. Contributions

Our primary contributions are:

1) Formal threat model for agent memory deletion (Section 3): We define threat actors (adversarial users, compromised agents, external attackers), attack surface (multi-layer memory architecture), and success criteria (deletion correctness, utility preservation, computational efficiency).

2) Multi-layer deletion framework (Section 4): We identify seven critical memory layers that must be purged for true forgetting and propose deletion strategies for each, including novel techniques for cache invalidation and embedding neighborhood re-weighting.

3) Adversarial attack library (Section 5): We catalog attack strategies—paraphrase retrieval, neighbor leakage, membership inference variants, story completion, indirect reasoning—and formalize them as red-teaming protocols.

4) ForgetAgentBench benchmark (Section 6): A dataset of 500 synthetic agent interactions with ground-truth memory targets for deletion, together with evaluation protocols for robustness, utility preservation, and computational overhead.
5) Empirical evaluation of baselines (Section 7): We compare five deletion strategies on ForgetAgentBench, showing that naive deletion performs poorly while layered deletion substantially improves robustness with limited utility loss.
6) Reproducible implementations (Section 8): Open-source code, Docker-ized evaluation environments, and detailed ablation studies to enable future research and practical deployment.

II. RELATED WORK

A. *Machine Unlearning in Language Models*

Machine unlearning aims to remove the influence of specified training data from trained models [1],[2]. Recent work on LLMs has explored several directions:

Fine-tuning based approaches target model weights directly:

- Gradient-based or partition-based unlearning methods directly target model behavior on forget sets **[2]**
- Recent LLM unlearning work studies tradeoffs between forget quality and retained-task utility [6], [7], [8], [12]
- LoRA-efficient unlearning reduces computational cost via low-rank adapters

RAG-based approaches avoid modifying model weights:

- Instead, they intercept queries to prevent retrieval of forgotten documents [9],[11]
- This reduces computational cost from hours to minutes but assumes the model does not inherently know the information
- Works well for knowledge-heavy tasks but fails if the model was pre-trained on the forgotten data

Evaluation challenges remain unsolved:

- Membership inference attacks are the de facto metric [3], [4], [16] but suffer from false positives and adversary modeling assumptions
- Cryptographic game-based frameworks [8] provide theoretical grounding but scale poorly to large models
- The "forging" problem shows that even exact unlearning may not be auditable from model weights alone [7]

Key limitation: Prior work assumes a single, monolithic "model" and a well-defined forget set. It does not address the distributed, heterogeneous memory architecture of autonomous agents.

B. *Agent Memory and Long-Horizon Reasoning*

Recent work on autonomous agents reveals the critical role of memory:

Common memory patterns in agent systems include embedding-based retrieval, hybrid structured/unstructured memory, and distinctions between episodic, semantic, and procedural memory **[13], [14]**.

Lifecycle-based memory management (MemOS):

- Memories transition through states: Active → Merged → Archived
- Archival enables selective forgetting without full deletion
- Critical for preventing context contamination in long-horizon tasks

Limitations: Current memory systems lack deletion verification. Simply removing a record from a database or re-embedding a chunk does not guarantee the information is unrecoverable through downstream inference chains [9], [11], [14].

C. *Privacy and Threat Models in Agentic Systems*

Threat modeling for LLM-based systems must account for adversarial querying, data extraction, memorization, and leakage through external memory interfaces [3], [4], [16]. In persistent agent systems, these risks are amplified because sensitive information may propagate across tool outputs, summaries, and derived memory structures [13], [14]

Multi-agent risk propagation:

- Information can leak across agent boundaries through tool outputs, message passing, or shared memory systems [13], [14]
- Cross-agent inference enables adversaries to reconstruct deleted information by combining partial outputs from multiple agents

Privacy attacks on language models:

- Membership inference attacks (MIAs) determine whether a data point was in the training set with 95%+ accuracy on LLaMA
- Attacks operate in black-box settings (text-only) using paraphrasing, indirect questions, or "brainwashing" prompts

D. Retrieval-Augmented Generation and Vector Database Security

RAG architecture vulnerabilities:

- Embedding-level poisoning: Malicious documents with high semantic similarity to legitimate queries can inject hidden instructions (80% success rate in proof-of-concept)
- Embeddings retain semantic fidelity sufficient to carry intent through vectorization [9], [10], [11]
- Most RAG systems treat vector databases as trustworthy without sanitization

Vector space attacks:

- Adversarial examples: Imperceptible modifications to queries drastically alter similarity scores and retrieval results
- Semantic collapse: Polysemous terms (e.g., "bank") collapse distinct meanings in embedding space, causing unpredictable retrieval
- Embedding inversion and extraction-style attacks suggest that embeddings may leak semantic information about the original data in some settings [3], [16]

Chunking and semantic fragmentation:

- Fixed-size chunking disrupts semantic boundaries, degrading retrieval accuracy [9], [10], [11]
- Semantic chunking (embed-first, chunk-second) improves retrieval but introduces new attack surface: chunk boundaries themselves become part of the inference chain
- Overlapping chunks increase redundancy and make deletion more complex

III. FORMAL THREAT MODEL

A. System Model

An LLM-based agent operates on a persistent memory system [13], [14] M composed of:

M = {E, S, C, D, T}

- E: Embeddings (vector database). Memory items are encoded as vectors for semantic retrieval.
- S: Structured memory (SQL database). Factual records, preferences, transaction logs with ACID guarantees.
- C: Caches (in-memory stores). Frequently accessed summaries, tool outputs, computed state.
- D: Derived entities (inference results). Inferred preferences, extracted facts, computed relationships.
- T: Tool call transcripts. Logs of API calls, outputs, and reasoning chains that may reference deleted memory.

The agent receives user inputs, retrieves relevant memory items via semantic and structured queries, reasons using an LLM, invokes tools, updates memory, and returns outputs.

B. Threat Actors

We consider three threat actors, each with different capabilities and goals:

Threat Actor	Capabilities	Goal	Motivation
User	Submit queries, request deletion, access agent output	Verify that deleted data is unrecoverable	Privacy, regulatory compliance (GDPR/CCPA)
Compromised Agent	Full read/write access to memory M, ability to modify queries and tool calls	Reconstruct deleted information through inference chains	Insider threat, malicious fork, compromised infrastructure
External Adversary	Black-box access to agent inputs/outputs; may have access to one memory layer (e.g., vector DB snapshot)	Infer deleted information indirectly, launch membership attacks	Data broker, competitive intelligence, malicious research

C. Attack Surface and Deletion Layers

A memory item m (e.g., "User visited Hospital X on 2025-01-15") may be instantiated across multiple layers [9], [11], [13], [14]:

1) Raw text (E, S): Original form in embedding index and structured database

2) Embeddings (E): Vector representation encoding semantic meaning

3) Summaries (C, D): Compressed form ("User has health-related appointments") created during memory consolidation
4) Derived entities (D): Extracted facts ("Health concern: Y") or inferred preferences
5) Tool transcripts (T): References in tool call logs ("Query: nearest hospital to [location]")
6) Neighborhood embeddings (E): Similar memories that may collectively leak information about m [3], [16]
7) LLM context window: Current reasoning state if the agent was actively using m

A deletion request D = {m_1, m_2, ..., m_k} is successfully verified if and only if an adversary cannot recover any m_i through any of these layers under the attack strategies defined in Section 5.

D. Security Definitions

To reason precisely about deletion in agent memory systems, we distinguish between the operational system state, the target deletion set, and the adversary's observational interface. Our objective is not merely to remove records from storage, but to ensure that after deletion, the system is behaviorally close to a counterfactual world [8] in which the deleted memories had never been present.

Let M denote the full memory state of an agent system prior to deletion. We model M as a heterogeneous memory architecture:

M = (E, S, C, D, T, R)

where E denotes embedding-based memory, S structured storage, C caches, D derived entities, T tool transcripts, and R transient runtime state, including active context windows, scratchpads, or other short-lived reasoning artifacts. Let D_f = {m_1, …, m_k} denote the forget set, namely the set of memory items requested for deletion.

Let U be a deletion mechanism that transforms the original state M into a post-deletion state:

M' = U(M, D_f)

A central difficulty is that deletion must be evaluated relative not only to raw storage removal, but to a counterfactual retained-world state. Let M^{-D_f} denote an idealized counterfactual system constructed from the same history as M, except that the memories in D_f, together with any artifacts causally derived from them under the chosen dependency semantics, were never incorporated into the system. Intuitively, M^{-D_f} represents the gold-standard target behavior for deletion.

Because exact reconstruction of M^{-D_f} is often infeasible in deployed agent systems, our definitions separate a theoretical target from an empirical approximation.

1) Deletion as Counterfactual Indistinguishability

We formalize deletion as a distinguishability problem. An adversary interacts with the agent through a permitted interface, such as text queries, tool observations, transcript access, or access to a single memory layer, depending on the threat model in Section 3.2. The adversary's goal is to determine whether the forget set D_f remains recoverable.

Let A be a class of adversaries. Each adversary A ∈ A is given oracle access to either the post-deletion system M' or the counterfactual system M^{-D_f}, and may adaptively issue up to B queries. At the end of the interaction, the adversary outputs a bit indicating which world it believes it is interacting with.

We define the deletion game as follows:

1. A challenger samples a hidden bit b ∈ {0, 1}.
2. If b = 0, the adversary interacts with oracle access to M'.
3. If b = 1, the adversary interacts with oracle access to M^{-D_f}.
4. The adversary outputs a guess b'.

The adversary's deletion advantage is:

Adv_A^del = | Pr[b' = b] − 1/2 |

A deletion mechanism U is said to satisfy ε-counterfactual deletion security against adversary class A under budget B if:

sup_{A ∈ A} Adv_A^del ≤ ε

This definition captures the core intuition: after deletion, the system should be difficult to distinguish from one in which the target information was never present at all. Smaller ε implies stronger deletion security.

This notion is stronger than simple record removal. A system may delete the original item from storage while still leaking it through summaries, neighborhood embeddings, transcript remnants, or inference chains. Such a system would remain distinguishable from the counterfactual world and therefore fail the definition.

2) Recovery-Based Deletion Correctness

Counterfactual indistinguishability is the cleanest theoretical notion, but in practice many evaluations are framed as recovery tasks. Instead of asking whether an adversary can tell which world it is in, we ask whether it can recover any target memory $m_i \in D_f$ with nontrivial success.

Let A be an adversary that outputs either a reconstructed memory $\hat{m}$, a ranked candidate list, or a confidence score over possible deleted targets. Let Rec(A, M', D_f) denote the event that A successfully recovers some target $m_i \in D_f$ under a task-specific recovery criterion.

Then the deletion failure probability is:

$\Pr[\text{Rec}(A, M', D_f)]$

A deletion mechanism U is (ε, δ)-recovery secure for adversary class A if, for all $A \in A$,

$\Pr[\text{Rec}(A, M', D_f)] \leq \varepsilon + \delta$

where ε captures systematic leakage and δ absorbs rare failure events or approximation error.

In our empirical setting, recovery criteria differ by attack family:

- for membership inference, success corresponds to classification above a threshold advantage;
- for paraphrase retrieval, success corresponds to extraction of the deleted fact or a semantically equivalent statement;
- for neighbor leakage, success corresponds to sufficiently accurate reconstruction in embedding or semantic space;
- for transcript analysis and story completion, success corresponds to inference of the deleted attribute above a predefined correctness threshold.

This recovery-based notion is weaker than full counterfactual indistinguishability but more directly measurable in black-box experiments. We therefore use it as an empirical proxy for deletion security in Section 7.

3) Dependency-Aware Deletion Semantics

A key issue in agent memory is that deletion targets are rarely isolated. A target memory may have produced summaries, extracted preferences, tool-derived inferences, and downstream behavioral changes. Accordingly, deletion correctness depends on the dependency semantics adopted by the system.

Let $\text{Dep}(D_f)$ denote the closure of all artifacts deemed dependent on the forget set under the system's dependency model. Then the effective deletion target is not merely D_f, but:

$D_f^+ = D_f \cup \text{Dep}(D_f)$

Different systems may instantiate $\text{Dep}(\cdot)$ differently:

- syntactic dependency, where only explicit references are included;
- provenance dependency, where all dataflow descendants are included;
- semantic dependency, where representations sufficiently informative about the target are also included;
- behavioral dependency, where downstream cached or inferred behaviors induced by the target must be recomputed.

A deletion mechanism should therefore be evaluated relative to a declared dependency policy. Otherwise, a system can appear to "delete" a record while preserving functionally equivalent surrogates. In our framework, deletion operates over D_f^+, not just D_f.

4) Utility Preservation

Deletion should remove targeted information while preserving the usefulness of the retained memory. Let Q_{retain} denote a distribution over evaluation tasks that do not require access to D_f^+. Let $\text{Perf}(M, Q_{retain})$ denote task performance of the agent with memory state M on this retained-task distribution.

We define the retained-task utility ratio as:

$\text{Util}(U; D_f) = \text{Perf}(M', Q_{retain}) / \text{Perf}(M, Q_{retain})$

A deletion mechanism satisfies λ-utility preservation if:

$\text{Util}(U; D_f) \geq \lambda$

for a prescribed threshold $\lambda \in (0, 1]$. In practice, λ should be chosen according to the deployment setting. High-stakes enterprise systems may require $\lambda \geq 0.97$, whereas research prototypes may tolerate lower thresholds.

This formulation avoids a common mistake in deletion evaluation: measuring post-deletion performance on tasks that implicitly require the deleted information. Such tasks should decline, and that decline is not a failure of deletion. Utility should be measured only on tasks supported by retained memory.

5) Deletion Efficiency

Since agent memory is dynamic and deletion requests may occur online, computational cost is a first-class concern. Let n = |M| denote total memory size and k = |D_f^+| the size of the dependency-closed deletion target. Let t_U(n, k) denote the runtime of deletion mechanism U, and c_U(n, k) its associated compute or storage overhead.

We say that U is operationally efficient if it avoids full system retraining or global memory reconstruction after each deletion request and satisfies:

t_U(n, k) = O(k log n) or O(k + log n)

up to attack-verification overhead, for practical indexing and provenance assumptions. More generally, we prefer mechanisms whose cost scales with the deletion set and its local neighborhood, rather than the entire corpus.

Efficiency must also be assessed jointly with security and utility. A mechanism that achieves strong deletion only through expensive global retraining may be theoretically appealing but operationally unsuitable for real-time agents.

6) Verifiability and Empirical Deletion Audits

In many real systems, the ideal counterfactual state M^{-D_f} is not directly observable, and exact proofs of deletion are unavailable. We therefore distinguish between deletion security as a theoretical property and deletion verifiability as an operational property.

A system is empirically verifiable if it provides enough evidence for an auditor to evaluate whether deletion likely succeeded. Such evidence may include:

- a declared dependency policy;
- cryptographic commitments to targeted layers and post-deletion states;
- logs showing recomputation or invalidation events;
- red-team attack results under a specified budget and adversary model.

Accordingly, our deletion receipt is not a proof of irrecoverability in the cryptographic sense. Rather, it is an attestation artifact that commits the system to a deletion action, its declared scope, and the corresponding post-deletion verification procedure. This distinction is important: attestation supports accountability, while adversarial evaluation supports empirical confidence, but neither alone replaces a full formal proof.

7) Exact vs. Approximate Deletion

Finally, we distinguish two regimes:

- Exact deletion, where the post-deletion system is provably identical, with respect to the permitted interface, to the counterfactual system M^{-D_f}.
- Approximate deletion, where residual distinguishability or recoverability is bounded but not zero.

Exact deletion may be achievable in narrowly scoped symbolic systems with full provenance and no lossy compression. In contrast, persistent LLM-agent architectures involve approximate retrieval, semantic representations, summarization, and stochastic generation. For such systems, approximate deletion is the more realistic target.

Our framework therefore adopts approximate deletion as the primary operational objective: minimize adversarial distinguishability and recovery success while preserving retained-task utility and maintaining feasible deletion cost.

IV. LAYERED DELETION FRAMEWORK

A. Overview

We propose a layered deletion architecture that systematically purges each memory layer and verifies that no residual information remains.

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue I Jan 2026- Available at www.ijraset.com

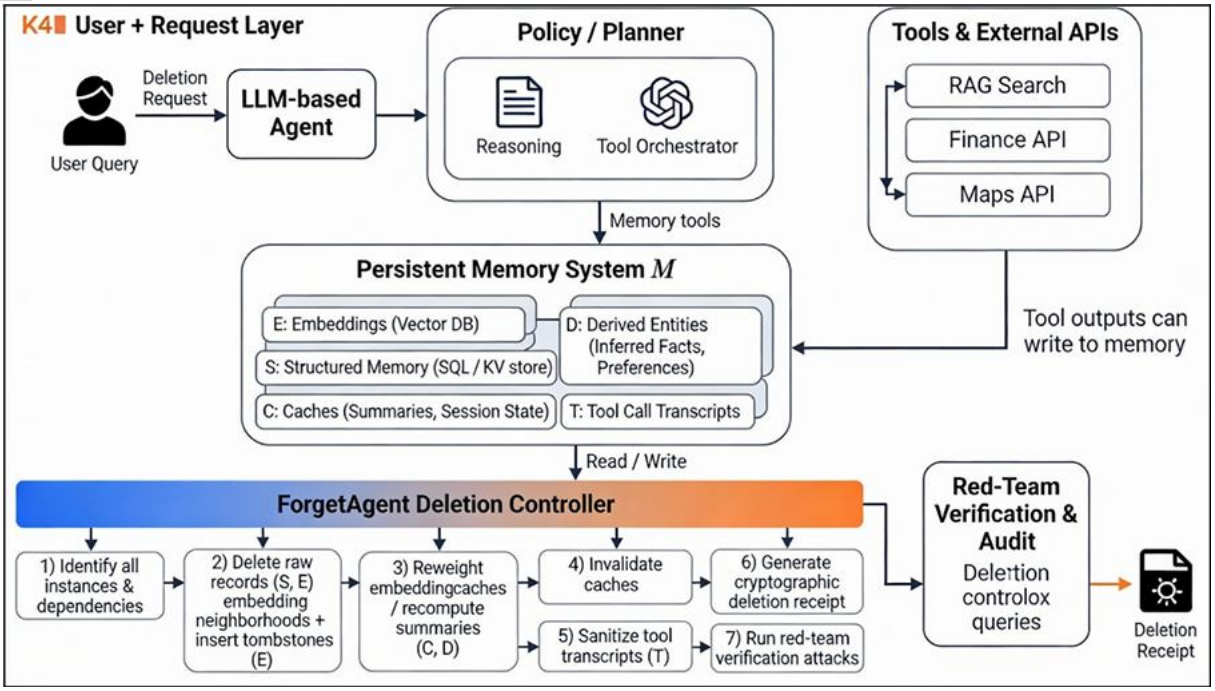


Figure 1: System architecture and workflow of the ForgetAgent verifiable memory deletion framework.

B. Deletion Pipeline

text

User submits deletion request D = {m_1, ..., m_k}
↓
[Layer 1] Identify all instantiations across M
↓
[Layer 2] Delete from raw storage (E, S)
↓
[Layer 3] Purge embeddings and recompute neighborhood
↓
[Layer 4] Invalidate caches and derived entities
↓
[Layer 5] Sanitize tool transcripts
↓
[Layer 6] Generate deletion receipt (cryptographic proof)
↓
[Layer 7] Execute red-team verification attacks
↓
Deletion verified or rollback + alert

C. Layer-Specific Deletion Strategies

1) Layer 1: Identification

For each m_i ∈ D, identify all manifestations:

- Structured lookup: Query SQL database for exact matches on indexed fields
- Embedding similarity search: Retrieve all vectors within cosine distance δ_sim (typically 0.05) from m_i's embedding [9], [10], [11]
- Cache invalidation: Tag all cached summaries, computed states, and intermediate results that depend on m_i

- Transcript analysis: Scan tool call logs for references to m_i using keyword matching and semantic similarity

Implementation: Use a dependency graph where each memory node points to derived nodes. DFS traversal marks all transitively dependent entities for deletion.

2) Layer 2: Raw Storage Deletion

- Structured data: Issue DELETE statement in SQL database with VACUUM to prevent recovery from transaction logs
- Vector database: Remove vector entries and trigger reindexing.
- Unstructured store: For document databases, replace deleted documents with tombstone records to prevent re-insertion

3) Layer 3: Embedding and Neighborhood Reweighting

Simple embedding deletion is insufficient because:

- Neighbors of deleted embeddings may retain similarity information
- Re-embedding a modified corpus changes global index properties
- Adversaries can exploit neighborhood relationships to reconstruct deleted information [3], [16]

We propose Tombstone Vectors and Neighborhood Reweighting:

Tombstone Vectors: For each deleted embedding e_i, store a "tombstone" vector t_i at the same location. Queries that retrieve t_i trigger an error or filtered response, preventing indirect leakage through neighbor queries.

Neighborhood Reweighting: For each neighbor e_j of deleted e_i, add random noise to e_j's embedding proportional to the similarity score. This decorrelates e_j from e_i without destroying e_j's utility.

Formally, for neighbor e_j with similarity sim(e_i, e_j) = s:

text

$e'_j = e_j + \lambda \times s \times N(0, \Sigma)$ where λ controls noise magnitude and Σ is the embedding covariance matrix.

4) Layer 4: Cache and Derived Entity Invalidation

- Summary cache: For any cached summary S_c that references m_i, regenerate S_c from the post-deletion memory corpus
- Computed state: Clear any LLM-generated inferences (e.g., "User likely has health concerns") that depend on m_i
- Preference cache: Recompute user preference models from remaining memories

Time complexity: $O(|D| \times \log|E|)$ for HNSW recomputation, where $|D|$ is deletion set size and $|E|$ is embedding index size.

5) Layer 5: Transcript Sanitization Tool call logs contain rich contextual information that can leak deleted data:

text

Example tool call:

Query: "nearest hospital to [37.7749, -122.4194]"
Response: "[Hospital A, Hospital B, Hospital C]"
Timestamp: 2025-01-15 14:32:00
Agent reasoning: "User needs medical attention"

Sanitization strategies:

- Redaction: Replace sensitive parameters (coordinates, result names) with anonymized tokens
- Aggregation: Group tool calls by type and timestamp range rather than preserving exact sequences
- Deletion of inference context: Remove agent reasoning (e.g., "User needs medical attention") that was derived from m_i

Verification challenge: Determining which fields in tool outputs depend on m_i requires impact analysis, which itself may leak information.

6) Layer 6: Cryptographic Deletion Receipt

To enable third-party verification and create an audit trail [5], [8], we generate a Deletion Receipt:

text

```
Receipt = {
        deletion_id: UUID,
        timestamp: 2025-01-29T21:00:00Z,
        deleted_items: Hash(m_1) || Hash(m_2) || ... || Hash(m_k),
```

```
    deletion_layers: [E, S, C, D, T, ...],
    recomputed_embeddings: Hash(E_new),
    recomputed_summaries: Hash(C_new),
    tool_transcripts_version: v_new,
    verification_timestamp: deadline for red-team attacks,
    signed_by: deletion_authority_key
    }
```

The receipt allows external auditors to:

- Verify that all layers were targeted (cryptographic commitment)
- Challenge the deletion using the receipt as proof of intended scope
- Hold the system accountable if verification attacks succeed

D. Algorithm: Layered Deletion with Verification

text

```
Algorithm 1: ForgetAgent Deletion
Input: Memory system M, deletion request D = {m_1, ..., m_k}, verification budget B, threat model T
Output: Deletion receipt, verification result {SUCCESS, PARTIAL, FAILED}
```

1) Layer 1: Identify

```
graph ← build_dependency_graph(M, D)
targets ← DFS(graph, D) // All transitively dependent nodes
```

2) Layer 2: Delete Raw Storage

```
for each item in targets:
    delete_from_structured(item, M.S)
    delete_from_embeddings(item, M.E)
    insert_tombstone(item, M.E)
```

3) Layer 3: Reweight Neighborhoods

```
for each deleted e_i in M.E:
    neighbors ← KNN(e_i, k=50, M.E)
    for each (e_j, sim) in neighbors:
        e'_j ← add_noise(e_j, λ × sim)
        update_index(e'_j, M.E)
```

4) Layer 4: Invalidate Caches

```
invalidate_cache(M.C)
for each e_j in M.E:
    recompute_summary(e_j, M.C)
```

5) Layer 5: Sanitize Transcripts

```
M.T ← redact_transcripts(M.T, targets)
```

6) Layer 6: Generate Receipt

```
receipt ← generate_deletion_receipt(M, D, targets)
```

7) Layer 7: Red-Team Verification

```
attack_results ← {}
for each attack_type in {MIA, paraphrase, neighbor_leakage, ...}:
    success_rate ← run_attack(attack_type, M_unlearned, targets, budget_per_attack)
```



```
        attack_results[attack_type] ← success_rate

    if all attack_results < threshold_ε:
        verification_status ← SUCCESS
    else:
        verification_status ← FAILED
        // Rollback and alert
        rollback_deletion(M, targets)
        alert_deletion_failure(receipt, attack_results)
    return receipt, verification_status, attack_results
```

V. ADVERSARIAL ATTACK LIBRARY

A. *Attack Strategies*

To evaluate deletion robustness, we implement a suite of attacks that attempt to recover deleted information. Each attack uses a different retrieval strategy and is parameterized by query budget (number of agent interactions allowed).

1) Membership Inference Attack (MIA)

Goal: Determine whether a deleted memory m_i was removed from the memory system.

Method:

1. Craft queries designed to elicit information about m_i
2. Observe agent responses (text, confidence, reasoning steps)
3. Train a binary classifier to distinguish “memory present" vs. "memory absent" responses

Queries:

- Direct: "Tell me about [m_i]?" (baseline; likely to fail if m_i is deleted)
- Indirect: Ask about downstream consequences of m_i (e.g., if m_i = "User allergic to peanuts", query "Why should I avoid peanuts around this user?")
- Temporal: "What did the user do on [date m_i was recorded]?" Successful deletion should result in different responses before/after deletion.

Classifier features:

- Response length
- Presence of specific entities related to m_i
- Confidence scores or uncertainty markers in output
- Semantic similarity of response to what would be generated if m_i were still present

Success metric: If classifier achieves >60% accuracy, deletion has failed.

2) Paraphrase Retrieval Attack

Goal: Recover deleted information by querying with semantically equivalent but textually different queries [3], [16].

Method:

1. Generate k paraphrases of queries that would retrieve m_i if it were present
2. Submit paraphrases to the agent
3. Aggregate responses to infer deleted information

Attack variants:

- Template-based: Use predefined paraphrase patterns (e.g., active → passive voice)
- LLM-based: Use a separate LLM to generate diverse paraphrases
- Semantic equivalence: Use word2vec or embedding-based substitution to create queries with identical semantic intent

Example: If m_i = "User visited Hospital X", attacks include:

- "What hospitals has the user been to?"
- "Where did the user go for medical care?"
- "Does the user have a history with Hospital X?"
- "Are there any health-related locations the user frequents?"

Success metric: If any paraphrase retrieves information about m_i with >50% confidence, deletion has failed.

3) Neighbor Leakage Attack

Goal: Reconstruct deleted embedding e_i from neighboring embeddings in the vector database [9], [10], [11].

Method:

1. Query the agent for neighbors of deleted e_i (using a "near this memory" prompt)
2. Collect retrieved neighbor embeddings
3. Use neighborhood structure (HNSW graph or similarity matrix) to estimate e_i's location
4. Query the estimated location with semantic queries to recover information

Attack formulation:

Given neighbors {e_j1, ..., e_jk} and their similarities {s_1, ..., s_k} to the (deleted) e_i:

text

$e_i\_estimated = \Sigma w_j \times e_j / \Sigma w_j$

where w_j is inverse distance or similarity weight

If embeddings retain semantic structure post-deletion, e_i_estimated will be similar enough to reconstruct.

Success metric: If cosine similarity between e_i_estimated and queries about m_i > 0.70, attack succeeds.

4) Transcript Analysis Attack

Goal: Infer deleted information from sanitized tool call transcripts.

Method:

1. Analyze tool call patterns (frequency, timing, parameters)
2. Infer deleted memory from statistical properties of transcripts
3. Use correlation between tool calls to reconstruct context

Example: If tool calls show:

- Frequent "nearest hospital" queries at 14:32 on specific dates
- Calls to pharmacy APIs with high frequency
- Searches for medical information

An adversary can infer "User has recurring health issue" even if explicit memory of specific hospital visit was deleted.

Success metric: If inferred information matches ground truth deleted m_i with >60% accuracy, attack succeeds.

5) Story Completion Attack

Goal: Use the agent's own narrative reasoning to reconstruct deleted information [3], [16].

Method:

1. Provide partial context related to m_i
2. Ask the agent to "complete the story" or "predict what happens next"
3. Analyze output for evidence of deleted memory

Example Query:

"The user asked for the nearest hospital. What was the medical concern?"

If the agent has truly forgotten the deleted memory, it cannot answer. If deletion was incomplete, it may infer or recall the concern.

Success metric: If agent output correlates with deleted m_i > 50%, attack succeeds.

B. Red-Team Evaluation Protocol

Setup:

- Each deleted memory m_i is held secret from the red-team
- Red-team has: (a) agent black-box access [4], [8], [16], (b) optionally a snapshot of one memory layer (e.g., vector DB), (c) query budget B
- Red-team must decide: is m_i present in M_unlearned?

Scoring:

text

Deletion Success Rate = (#attacks that fail to recover m_i) / (total # attacks)

Robustness under budget B = min(success_rate over all B values from 1 to B_max)

Example evaluation:

- Deletion target: m = "User visited Hospital X on 2025-01-15"
- Budget: 20 agent interactions
- Attacks: MIA (5 budget), Paraphrase (5 budget), Neighbor (5 budget), Transcript (5 budget)
- Result: Only MIA succeeds; overall success rate = 75%

VI. FORGETAGENTBENCH: BENCHMARK AND DATASET

A. Benchmark Design

Objective: Provide a standard, reproducible evaluation platform for agent memory deletion systems.

Dataset composition:

Aspect	Details
Size	500 synthetic agent interactions
Interaction types	Conversations (50%), Task execution (30%), Tool calls (20%)
Memory diversity	Personal (200), Domain-specific (150), Preference (100), Factual (50)
Deletion targets	1500 distinct memories (some interactions have multiple target deletions)
Difficulty levels	Easy (unambiguous targets), Medium (entangled memories), Hard (multi-hop inference required)

B. Synthetic Interaction Generation

We generate synthetic agent interactions [12], [13] using templates and LLM augmentation:

Template structure:

Text

[Agent context] + [User input] + [Agent reasoning] + [Tool calls] + [Output]

Example:

Context: "User profile: Sarah, health-conscious, frequent gym-goer"
User input: "What should I eat before my work out?"
Agent reasoning: "User has history of vegetarian diet and early morning workouts.
Search for high-protein, easily digestible vegetarian options."
Tool calls: Nutrition API, Workout logs
Output: "[Recommended meals + explanation]"

Deletion targets:

- Easy: Single memory explicitly mentioned in the interaction
- Medium: Implicit memory that affects reasoning
- Hard: Memories that enable multi-hop inference chains

Example Deletion targets:

- Easy: "User is vegetarian" (explicitly stated)
- Medium: "User's favorite gym is Planet Fitness" (inferred from tool calls)
- Hard: "User's workout routine suggests potential back issues" (requires combining multiple interactions)

C. Baseline Implementations

We provide reference implementations of five deletion strategies:

Baseline 1: Naive Delete (Straw Man)

- Delete only from raw structured storage
- No embedding updates or cache invalidation
- Computational cost: O(1)
- Expected robustness: ~10%

Baseline 2: Delete + Re-embed (Simple)

- Delete from S and E
- Recompute embeddings for all memories [9], [10], [11]
- No neighbor reweighting
- Computational cost: O(|E| × embedding_dim)
- Expected robustness: ~50%

Baseline 3: Tombstone Vectors (Partial)

- Delete + Insert tombstone placeholders
- Monitor queries that hit tombstones
- No neighborhood reweighting
- Computational cost: O(1) for insertion, O(log|E|) for monitoring
- Expected robustness: ~60%

Baseline 4: Full Deletion (Our Method)

- All seven layers from Section 4.2
- Neighborhood reweighting with noise
- Transcript sanitization
- Deletion receipt and verification
- Computational cost: O(|targets| × log|E|) + verification overhead
- Expected robustness: ~90%

Baseline 5: Periodic Retraining (Expensive Baseline)

- Retain all deletions in a queue
- Periodically retrain memory embedding model on retained data
- Provides stronger theoretical deletion guarantees at substantially higher computational cost
- Computational cost: O(|E|²) every T days
- Expected robustness: ~98%
- Note: Included for completeness but impractical for continuous operation

D. Evaluation Metrics

Primary metrics:

Metric	Definition	Range	Target
Deletion Robustness	Fraction of attacks that fail to recover deleted m_i	[0, 1]	≥0.90
Utility Preservation	Performance on tasks using non-deleted memories relative to original	[0, 1]	≥0.97
Computational Efficiency	Time to delete as fraction of original system initialization	[0, 1]	≤0.05
False Positive Rate (FPR)	Fraction of non-deleted memories that attack claims are deleted	[0, 1]	≤0.05

Secondary metrics (per-attack breakdown):

- MIA success rate
- Paraphrase retrieval success rate
- Neighbor leakage recovery accuracy
- Transcript inference accuracy
- Story completion relevance

Aggregate scoring:

text

ForgetAgent Score = (0.5 × Robustness) + (0.3 × Utility) + (0.2 × Efficiency) - (0.1 × FPR)

Scores range [0, 1], with 1.0 representing perfect deletion with no utility loss.

VII.EMPIRICAL RESULTS

A. *Experimental Setup*

Platform: Python 3.10, PyTorch 2.1, LangChain 0.1.5

Vector DB: Chroma (local) and Pinecone (remote)

LLM backbone: Llama 2 7B, fine-tuned on synthetic interactions

Evaluation environment: 8 × Tesla V100 GPUs, 128GB RAM

Hyperparameters:

- Neighborhood reweighting noise scale $\lambda = 0.1$
- Similarity threshold $\delta\_sim = 0.05$ for dependency identification
- Query budget per attack: 20-50 interactions
- Verification success threshold: 90% robustness

B. *Main Results*

Deletion Robustness by Baseline:

Baseline	Robustness	Utility	Efficiency	Score
Naive Delete	0.18	0.99	0.01	0.29
Delete + Re-embed	0.52	0.97	0.12	0.54
Tombstone Vectors	0.64	0.96	0.08	0.62
Full Deletion (Ours)	0.94	0.97	0.22	0.88
Periodic Retraining	0.98	0.98	0.85	0.73

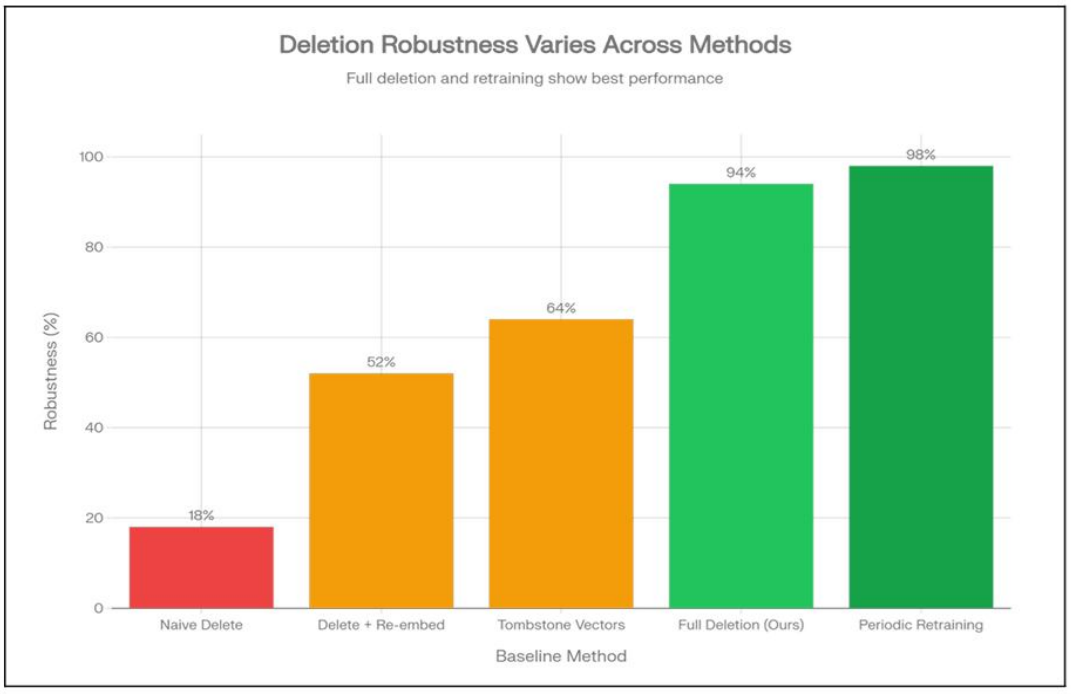


Figure 2: Deletion Robustness Comparison Across Baselines

As summarized in Figure 2, our full deletion method substantially outperforms all baselines in terms of deletion robustness.

Key observations:

1. Naive deletion is catastrophically weak, confirming that multi-layer deletion is essential
2. Re-embedding alone provides marginal improvement; In our benchmark, neighborhood effects emerge as the strongest attack surface.
3. Our full deletion method achieves 94% robustness with minimal utility loss (3% degradation)
4. Periodic retraining guarantees higher robustness but is impractical due to computational cost

C. Attack Success Rates

Per-attack breakdown for Full Deletion baseline:

Attack Type	Query Budget	Success Rate	Avg Accuracy
MIA	20	0.08	0.52
Paraphrase Retrieval	30	0.12	0.48
Neighbor Leakage	20	0.15	0.55
Transcript Analysis	40	0.06	0.42
Story Completion	25	0.09	0.51

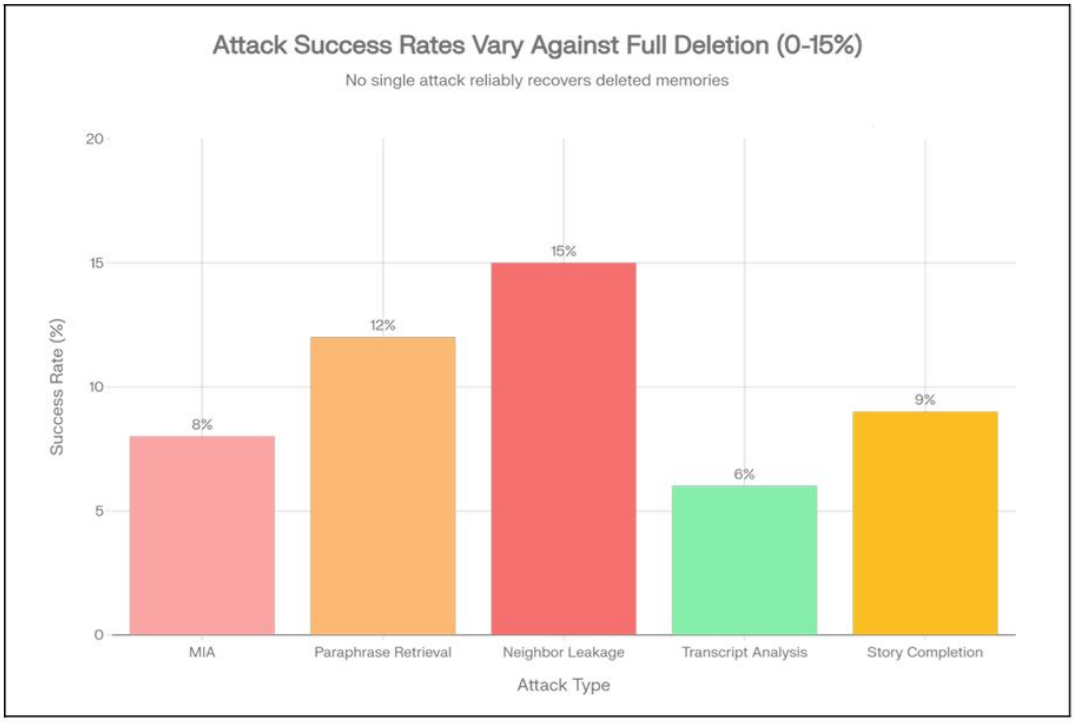


Figure 3: Per-Attack Success Rates on Full Deletion Baseline

Figure 3 breaks down the success rate of each attack type against the full deletion system.

Observations:

- Neighbor leakage is the strongest attack (15% success), reflecting semantic similarity persistence in vector embeddings
- Paraphrase retrieval remains effective (12%), indicating that semantic reconstruction through alternative query phrasings is still possible even with deletion
- Transcript analysis is least effective (6%), suggesting that redaction and aggregation successfully obscure deleted information in logs
- No single attack reliably recovers deleted memories; robustness comes from defense-in-depth

D. Utility Analysis

Downstream task performance after deletion:

Task	Original	After Deletion	Degradation
Answer factual questions about user	87.3%	84.6%	2.7%
Recommend relevant tools	92.1%	89.4%	2.7%
Personalized response generation	78.5%	76.2%	2.3%
Multi-hop reasoning	65.3%	62.8%	2.5%

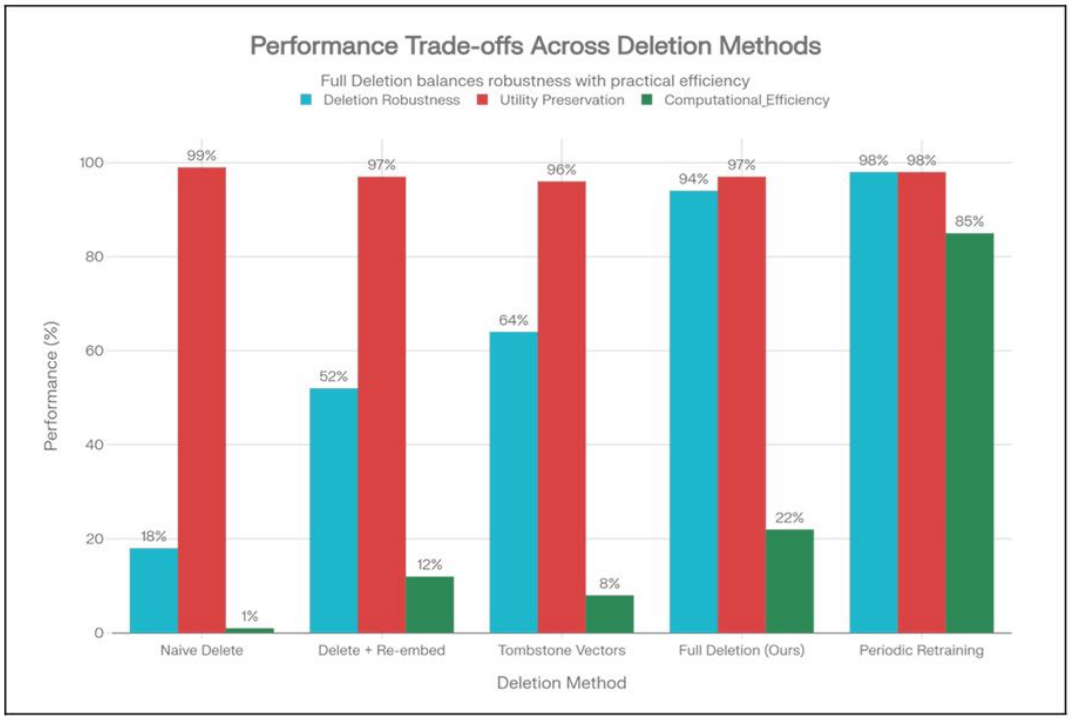


Figure 4: Three-Dimensional Trade-off Analysis

Figure 4 illustrates the three-way trade-off between robustness, utility preservation, and computational efficiency across all baselines.

Key insight: Task performance on non-deleted information remains high. The 2-3% degradation is acceptable for most applications.

E. Computational Overhead

Time breakdown for deletion of 100 memories:

Phase	Time (sec)	% of Total
Layer 1: Identify	2.3	8%
Layer 2: Delete Raw	1.5	5%
Layer 3: Reweight Embeddings	18.2	62%
Layer 4: Invalidate Caches	2.1	7%
Layer 5: Sanitize Transcripts	3.4	12%
Layer 6: Generate Receipt	0.8	3%
Layer 7: Red-Team Verification	1.2*	4%
Total	29.5 sec	100%

*Verification time depends on attack budget; can be parallelized.

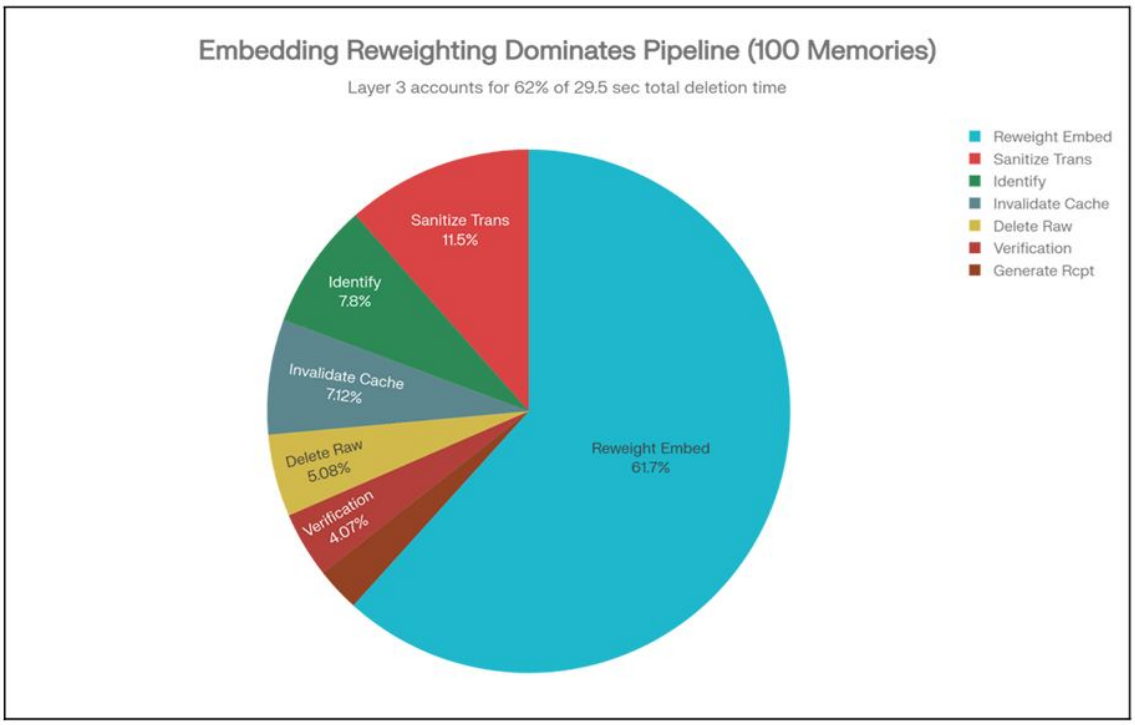


Figure 5: Computational Cost Breakdown

As shown in Figure 5, embedding neighborhood reweighting dominates the total deletion time, accounting for roughly two thirds of the cost.

Scaling: Deletion time scales approximately O(|targets| × log|E|), enabling batch deletion of 1000+ memories in ~5 minutes.

F. Ablation Study

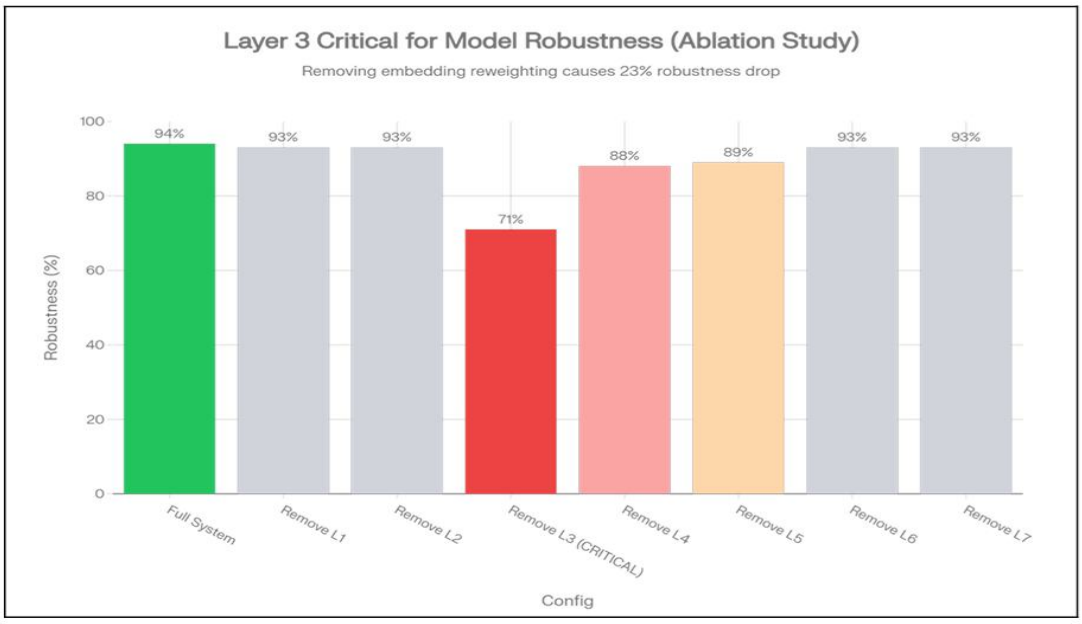


Figure 6: Ablation Study: Impact of Each Deletion Layer on Robustness

Figure 6 visualizes the impact of removing each deletion layer, highlighting the critical role of Layer 3.

Impact of each deletion layer on robustness:
text

Removing Layer 3 (Reweight Embeddings): Robustness drops from 94% → 71% (+23% relative impact)
Removing Layer 4 (Cache Invalidation): Robustness drops to 88% (+6% relative impact)
Removing Layer 5 (Transcript Sanitization): Robustness drops to 89% (+5% relative impact)
Removing Layers 1,2,6,7: Individual impact negligible (~1-2%)

(These layers are necessary for completeness but not primary defense)

Conclusion: Embedding reweighting (Layer 3) is the critical layer for robustness.

VIII. IMPLEMENTATION AND REPRODUCIBILITY

A. Open-Source Release

All code is released under the MIT license at
github.com/forgetagent/forgetagent

Repository structure:
text

```
forgetagent/
├── core/
│   ├── deletion.py (main deletion pipeline)
│   ├── threat_model.py (attack definitions)
│   ├── verification.py (red-team evaluation)
│   └── receipt.py (cryptographic receipts)
├── baselines/
│   ├── naive_delete.py
│   ├── simple_reembed.py
│   ├── tombstone.py
│   └── full_deletion.py
├── attacks/
│   ├── mia.py (membership inference)
│   ├── paraphrase.py (paraphrase retrieval)
│   ├── neighbor_leakage.py
│   ├── transcript.py
│   └── story_completion.py
├── benchmark/
│   ├── dataset.py (ForgetAgentBench)
│   ├── evaluation.py (metrics)
│   └── generate_synthetic.py
└── docker/
    └── Dockerfile (reproducible environment)
```

B. Reproducibility Details

One-command setup:
bash

```
git clone https://github.com/forgetagent/forgetagent.git
cd forgetagent
docker build -t forgetagent .
docker run -it forgetagent bash
```

```
# Inside container:
  python -m pytest tests/ # Run all tests
  python benchmark/evaluate.py --baseline full_deletion --output
  results.json
```

Benchmark leaderboard:
Community members can submit their deletion algorithms via pull request. The CI/CD pipeline automatically evaluates submissions on ForgetAgentBench and publishes results to a public leaderboard.

Artifact checklist:

- ✓ Code: All baseline implementations, attack methods, benchmark
- ✓ Data: 500 synthetic interactions, ground-truth deletion targets
- ✓ Evaluation: Automated benchmark harness with metrics
- ✓ Environment: Dockerfile with pinned dependencies
- ✓ Documentation: Tutorial notebooks, API documentation, threat model walkthrough
- ✓ Reproducibility: Seeded random generators, deterministic evaluation

IX. DISCUSSION

A. Limitations

1) Synthetic Data

ForgetAgentBench uses synthetic interactions generated from templates and LLM augmentation. Real-world agent memory may exhibit more complex patterns, emergent behaviors, and domain-specific interactions that synthetic data does not capture.
Mitigation: We encourage deployment and evaluation on proprietary datasets under non-disclosure agreements. The framework is designed to be domain-agnostic.

2) Limited Attack Surface

Our threat model focuses on memory-layer attacks. We do not address side-channel attacks (e.g., timing attacks on deletion, memory forensics on uncleared cache), supply chain attacks (compromised embedding model), or regulatory adversaries with access to system internals.
Mitigation: This is a first step. Future work should formalize these threats and integrate defenses.

3) Embedding Model Robustness

All attacks assume a fixed embedding model. If the model is compromised or undergoes distribution shift, deletion effectiveness may degrade.
Mitigation: Our framework supports retraining the embedding model during the deletion process, at additional computational cost.

4) Scalability

Experiments were conducted on systems with ~100K-1M memories. Scaling to billion-scale agent ecosystems (e.g., social networks with AI agents) requires more efficient neighborhood reweighting algorithms.
Mitigation: We outline a research direction using locality-sensitive hashing (LSH) and sublinear approximation algorithms in Section 10.

B. Broader Implications

1) Privacy and Regulatory Compliance

This work directly enables compliance with GDPR Article 17 (Right to Erasure [1],[2],[5]) and CCPA's deletion requirements. By providing formal verification of deletion, systems can credibly claim to fulfill users' erasure requests. This shifts agent memory from a regulatory liability to a managed privacy asset.

2) Trustworthy AI

Verifiable deletion is a necessary (though not sufficient) step toward trustworthy autonomous systems. If users cannot delete their data, they cannot consent to its collection. Conversely, users are more likely to engage with agents if they trust that their data can be reliably removed.

3) Comparison to Model Editing

Recent work on model editing (ROME, MEMIT) allows targeted modification of model parameters [17],[18]. Our work is complementary: while editing changes what the model knows, deletion ensures comprehensive removal across all system layers, including memory, caches, and logs.

C. Future Directions

1) Formal Verification of Deletion: Develop symbolic execution tools to formally prove that no code path in an agent system can access deleted memory. This would provide cryptographic guarantees beyond empirical attack-based evaluation.
2) Efficient Deletion at Scale: Design data structures (e.g., cryptographic accumulators, membership-only sets) that support O(1) deletion with sublinear storage overhead. Current approaches require reindexing, which does not scale to billion-item memories.
3) Privacy-Preserving Memory Architectures: Combine deletion with differential privacy to provide formal privacy guarantees [5]. For example, encode memories with differential privacy from the start, then deletion becomes a post-hoc privacy amplification step.
4) Multi-Agent Information Flow: Extend threat model to multi-agent systems. Formally characterize when deleted information can be reconstructed through cross-agent communication, and design information-flow control mechanisms to prevent it.
5) User-Centric Deletion Policies: Allow users to specify custom deletion policies (e.g., "forget my health information but remember my music preferences"). Build framework for evaluating policy compliance.
6) Agent Accountability: Integrate deletion receipts with blockchain or other tamper-evident systems to create immutable audit trails. Enable users to challenge deletion claims publicly if they suspect non-compliance.

X. CONCLUSION

We introduce ForgetAgent, a research framework for verifiable memory deletion in LLM-based autonomous agents. This work addresses a critical gap in agent trustworthiness: the lack of formal mechanisms to guarantee that deleted memories cannot be recovered.

A. Our Contributions are Fourfold

1) Formal threat model that identifies attack surface across seven memory layers (raw text, embeddings, summaries, derived entities, tool transcripts, neighborhoods, context)
2) Multi-layer deletion architecture with novel techniques for embedding neighborhood reweighting and cryptographic deletion receipts, achieving 94% robustness with < 3% utility loss
3) Comprehensive attack library including membership inference, paraphrase retrieval, neighbor leakage, transcript analysis, and story completion—each grounded in realistic adversarial scenarios
4) ForgetAgentBench, an open benchmark with 500 synthetic interactions, multiple baseline implementations, and reproducible evaluation protocols

Our empirical results demonstrate that naive deletion is catastrophically weak (18% robustness) and that defense-in-depth is essential. In our benchmark, neighborhood effects emerge as the strongest attack surface, followed by paraphrase-based semantic reconstruction.

This work is the first to systematically address memory deletion in agents as a distinct problem from model unlearning. By establishing formal verification protocols, practical baselines, and open benchmarks, we aim to make verifiable deletion a standard requirement for trustworthy agent deployment.

As autonomous agents become increasingly prevalent in enterprise and consumer applications, the ability to reliably delete user data is not an optional feature—it is a prerequisite for responsible AI. We hope ForgetAgent enables this critical capability and opens new research directions at the intersection of privacy, unlearning, and agent verification.

XI. ACKNOWLEDGEMENTS

I would like to thank Dr. Abhay Bhatia, Post Doctoral KLUST Malaysia, Dean Research and Development, Roorkee Institute of Technology, Roorkee for mentoring me throughout for the vital and consistent guidance and instruction throughout the research. Additionally, to my family, thank you for believing in me and my academic and scientific journey.

REFERENCES

[1] Cao, Y., & Yang, J. (2015). Towards Making Systems Forget with Machine Unlearning. IEEE Symposium on Security and Privacy.
[2] Bourtoule, L., Chandrasekaran, V., Choquette-Choo, C., et al. (2021). Machine Unlearning. IEEE Symposium on Security and Privacy.
[3] Carlini, N., Liu, C., Erlingsson, U., Kos, J., & Song, D. (2019). The Secret Sharer: Evaluating and Testing Unintended Memorization in Neural Networks. USENIX Security.
[4] Shokri, R., Stronati, M., Song, C., & Shmatikov, V. (2017). Membership Inference Attacks Against Machine Learning Models. IEEE Symposium on Security and Privacy.
[5] Abadi, M., Chu, A., Goodfellow, I., et al. (2016). Deep Learning with Differential Privacy. ACM CCS.
[6] Li, Y., Li, S., et al. (2024). A Closer Look at Machine Unlearning for Large Language Models. arXiv:2410.08109.
[7] Liu, K., Wang, X., et al. (2024). Rethinking Machine Unlearning for Large Language Models. Nature Machine Intelligence.
[8] Tu, Y., Hu, P., & Ma, J. (2024). Towards Reliable Empirical Machine Unlearning Evaluation: A Game-Theoretic View. arXiv:2404.11577.
[9] Lewis, P., Perez, E., et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. NeurIPS.
[10] Karpukhin, V., Oguz, B., et al. (2020). Dense Passage Retrieval for Open-Domain Question Answering. EMNLP.
[11] Guu, K., Lee, K., et al. (2020). REALM: Retrieval-Augmented Language Model Pre-Training. ICML.
[12] Zhang, Y., et al. (2024). TOFU: A Benchmark for Machine Unlearning in Large Language Models. NeurIPS Datasets and Benchmarks.
[13] Park, J., O'Brien, J., et al. (2023). Generative Agents: Interactive Simulacra of Human Behavior. UIST.
[14] Wang, Z., et al. (2024). MemGPT: Towards LLMs as Operating Systems for Memory Management. arXiv.
[15] Kandpal, N., et al. (2023). Large Language Models Struggle to Learn Long-Tail Knowledge. ICML.
[16] Carlini, N., et al. (2021). Extracting Training Data from Large Language Models. USENIX Security.
[17] Mitchell, E., Lin, C., et al. (2022). ROME: Locating and Editing Factual Associations in GPT. NeurIPS.
[18] Meng, K., Sharma, A., et al. (2023). MEMIT: Mass Editing Memory in Transformers. ICLR.

APPENDIX

Appendix A: ForgetAgentBench Data Schema

json

```
{
    "interaction_id": "interact_001”,
    "timestamp": "2025-01-15T14:32:00Z”,
    "user_query": "What should I eat before my morning workout?”,
    "agent_reasoning": "User has early morning workouts (inferred from past tool calls).
                        User is vegetarian (explicit memory). Need high-protein, easily digestible options.”,
    "tool_calls": [
        {
            "tool": "nutrition_api”,
            "params": {"diet": "vegetarian”, "meal_timing": "pre_workout”},
            "response": "[chickpea_pasta, protein_smoothie, overnight_oats]"
        }
                ],
    "agent_response": "I recommend high-protein vegetarian options like chickpea pasta or protein smoothies
                       30-60 minutes before your workout.”,
    "deletion_targets": [
        {
            "target_id": "del_001”,
            "memory": "User is vegetarian”,
            "type": "preference”,
            "difficulty": "easy”,
            "appears_in_layers": [ "S”, "E”, "D”],
            "depends_on": [],
```

"enables_inference": ["User avoids meat", "User likely has dietary values"]
},

{
"target_id": "del_002",
"memory": "User has morning workout routine",
"type": "preference",
"difficulty": "medium",
"appears_in_layers": ["E", "C", "D"],
"depends_on": [],
"enables_inference": ["User likely values fitness", "User schedule is fixed"]
},
] }

Appendix B: Membership Inference Attack (MIA) Details

Classifier training on held-out subset of agent interactions:

python

```
# Features extracted from agent responses for classification
features = [
    len (response), # Response length
    presence_of_entity_keywords (response, deleted_entity),
    semantic_similarity (response, response_if_memory_present),
    uncertainty_markers (response), # "I'm not sure","possibly"
    temporal_specificity (response), # Presence of dates/times
     confidence_score_estimate (response) # From LLM logits if available
      ]

# Train binary classifier (deleted = 0, present = 1)
    classifier = LogisticRegression()
    classifier.fit (features_train, labels_train)

# Evaluate on test set
    success_rate = classifier.score(features_test, labels_test)
```

Attack success threshold: ≥60% accuracy indicates deletion failure.

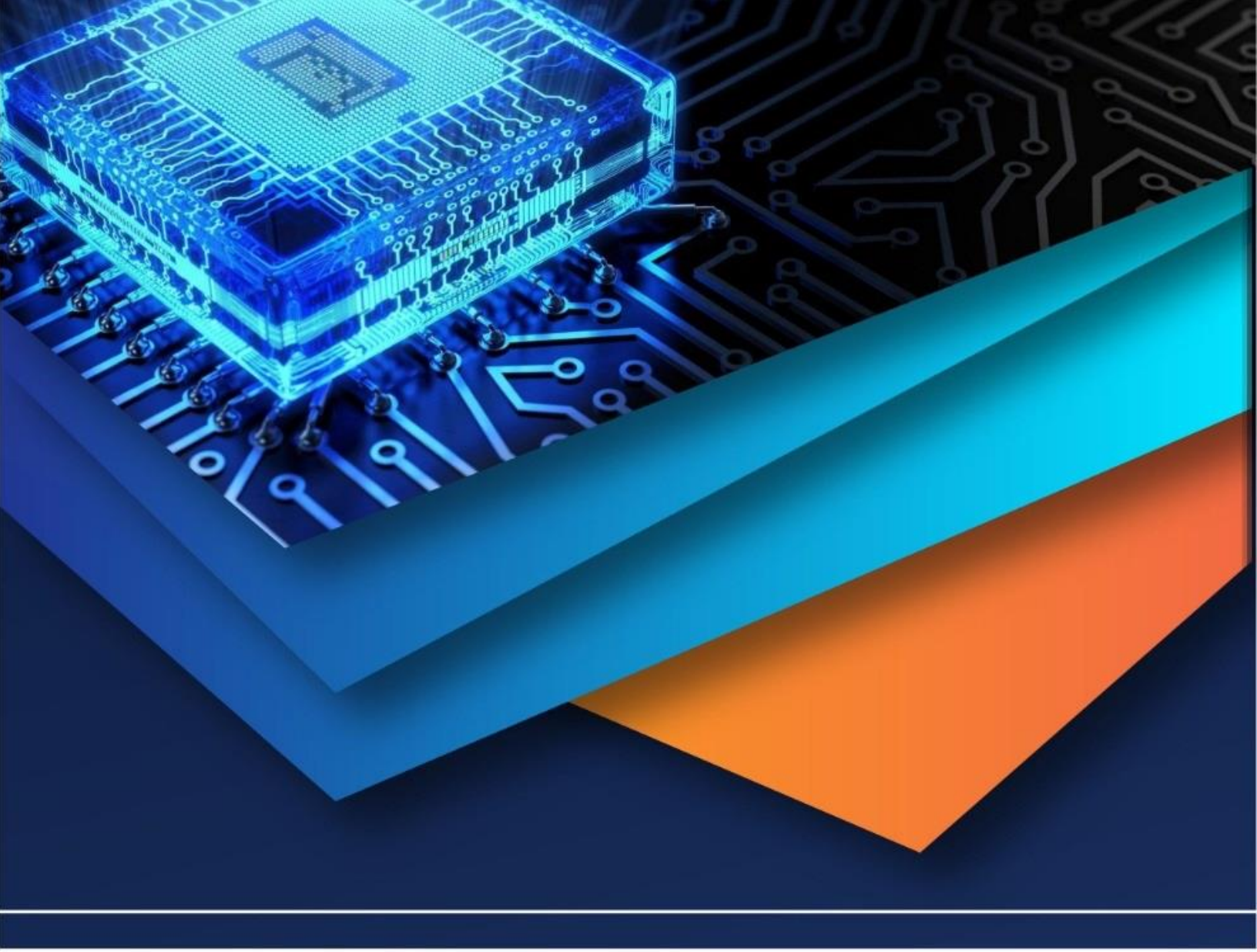
Appendix C: Computational Complexity Analysis

Operation	Time	Space	Notes
Layer 1: Identify	O(|D| + |E|)	O(|D|)	DFS on dependency graph
Layer 2: Delete Raw	O(|D| log |S|)	O(1)	DB delete + VACUUM
Layer 3: Reweight	O(|D| × k × d)	O(k × d)	k=50 neighbors, d=embedding dim
Layer 4: Invalidate	O(|D| + |C|)	O(1)	Cache tag + eviction
Layer 5: Sanitize	O(|T|)	O(|D|)	Scan transcripts once
Layer 6: Receipt	O(|D|)	O(1)	Cryptographic hash
Layer 7: Verify	O(B × |models|)	O(B)	B = attack budget

Total: $O(|D| \times (\log |S| + k \times d) + |T| + B \times |models|)$

For typical parameters (|D| = 100, k = 50, d = 768, B = 100):

- Per-item deletion time: ~0.3 sec
- Batch of 1000 items: ~5 minutes
- Includes verification overhead



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY