Text2Mem: A Unified Memory Operation Language for Memory Operating System

Anonymous ACL submission

Abstract

Large language model agents increasingly rely on memory to support long-horizon interaction, yet existing frameworks expose only a small set of low-level primitives and lack a formal, executable specification for memory control. As a result, higher-order operations such as promotion, consolidation, or lifecycle governance are missing or inconsistently implemented, leading to unpredictable behavior across systems. We introduce **Text2Mem**[1], a unified memory operation language that standardizes the translation of natural-language instructions into reliable execution. Text2Mem defines a compact and expressive operation set spanning encoding, storage, and retrieval, and represents each instruction as a schema-based contract with explicit fields and semantic invariants. Validated schemas are parsed into typed operation objects and executed through a unified pipeline that supports both a SQL reference backend and real memory frameworks, enabling safe, deterministic, and portable behavior across heterogeneous systems. We further outline the **Text2Mem Benchmark**, which decouples schema generation from backend execution to systematically evaluate planning accuracy and execution fidelity. Together, Text2Mem and its benchmark establish a standardized foundation for controllable and reproducible memory management in LLM-based agents.

1 Introduction

Large language model (LLM) agents (Zhao et al., 2025; Wang et al., 2024; Luo et al., 2025) are rapidly evolving from single-turn dialogue systems toward long-horizon agents capable of multi-session interaction and extended task execution. In this transition, memory becomes a central capability: it maintains consistent identity, accumulates user preferences, and provides contextual grounding across time, enabling persistent reasoning and personalized behavior (Yang et al., 2024; Wei et al., 2025; Li et al., 2025).

Despite its importance, current memory subsystems remain limited. Most frameworks expose only a small set of basic primitives such as *encode*, *retrieve*, and *delete*, while higher-level controls—including *merge*, *promote*, *demote*, *split*, *lock*, and *expire*—are often missing or inconsistently implemented (Packer et al., 2024; Chhikara et al., 2025). This incompleteness leads to poor portability across systems, limited compositionality for complex tasks, and restricted usability for managing memory lifecycles.

A second challenge is the lack of a formal and executable specification for memory operations. Natural-language instructions such as “archive outdated project notes” or “prioritize urgent reminders” are inherently underspecified in scope, duration, and action type. Without a schema that enforces required fields and normalizes parameters such as time and priority, memory commands cannot be executed deterministically, resulting in unpredictable behavior across systems.

To address these challenges, we introduce **Text2Mem**, a unified memory operation language for translating natural-language instructions into executable memory operations. Text2Mem defines a structured set of memory operations spanning encoding, storage, and retrieval, including higher-level controls such as promotion, consolidation, and lifecycle management, providing explicit support for operations that are often implicit or inconsistently handled in existing systems.

Rather than an ad-hoc memory framework, Text2Mem is designed as a language-level specification that defines the structure, scope, and required parameters of memory operations, while leaving their internal realization to specific backends. Each operation is represented by a formal schema with explicit fields and constraints, allowing underspecified natural-language commands to

[1] Code is available at `https://anonymous.4open.science/r/text2mem`

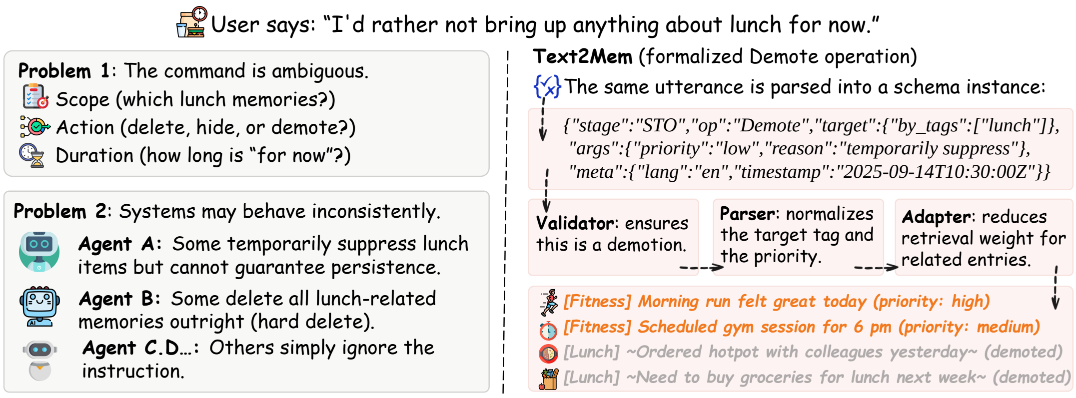

Figure 1: Ambiguity and inconsistency in current systems versus Text2Mem's formalized handling. **Left**: The natural language instruction "I'd rather not bring up anything about lunch for now" is underspecified, leading to inconsistent behaviors across agents. **Right**: Text2Mem resolves the ambiguity by instantiating a schema-based *Demote* operation with explicit arguments, enabling consistent execution across heterogeneous backends.

be normalized into well-formed operation instances prior to execution.

This paper makes the following contributions:

- We propose **Text2Mem**, the first unified memory operation language for LLM-based agents, defining a compact set of twelve operations with clear semantic boundaries across encoding, storage, and retrieval.
- We introduce a **schema-based specification** that formalizes memory operations through required fields, invariants, and typed parsing, enabling safe and deterministic execution.
- We demonstrate **backend portability** by executing the same typed operations on both a SQL-based reference backend and real memory frameworks with consistent behavior.
- We present the **Text2Mem Benchmark**, a two-layer evaluation suite that separately measures schema generation accuracy and execution consistency, providing a quantitative foundation for studying memory-centric reasoning in LLM agents.

2 Background and Motivation

2.1 Agent memory and operation frameworks

Prior work has explored augmenting LLM agents with diverse memory mechanisms. Human-inspired and functional approaches draw on cognitive theories, explicit working-memory structures, or behaviors such as note-taking and summarization to improve consolidation, recall efficiency, and durability (Gutierrez et al., 2024; Gutiérrez et al., 2025; Yang et al., 2024; Liang et al., 2024; Wei et al., 2025; Wu et al., 2023). In parallel, tool-based methods expose interfaces for editing or extending memory, including parameter-level knowledge modification and external memory modules that mitigate context-window limitations (Zhang et al., 2024; Xu et al., 2025b; Chhikara et al., 2025; Zhong et al., 2024; Rasmussen et al., 2025).

More system-oriented designs treat memory as a first-class operating component, exemplified by MemGPT, A-MEM, and MemOS (Packer et al., 2024; Xu et al., 2025a; Li et al., 2025). While these frameworks substantially advance practical memory manipulation, they remain fragmented: most rely on CRUD-style primitives or ad-hoc extensions, and lack a unified, semantically precise operation language to support higher-order controls such as prioritization, consolidation, and lifecycle governance.

2.2 Lessons from text-to-SQL

Text-to-SQL offers a direct parallel to memory control in LLM agents: underspecified natural language must be mapped to precise, executable operations. Early approaches translated utterances directly into SQL (Seq2SQL (Zhong et al., 2017)), while later models incorporated schema structure and constraints (RAT-SQL (Wang et al., 2020), UniSAr (Dou et al., 2022)). With large language models, both in-context learning and supervised fine-tuning have substantially improved robustness and accuracy (TA-SQL (Qu et al., 2024), SAFE-SQL (Lee et al., 2025), XiYan-SQL (Liu et al., 2025b), SQL-LLaMA (Lindorfer, 2023)) (Hong

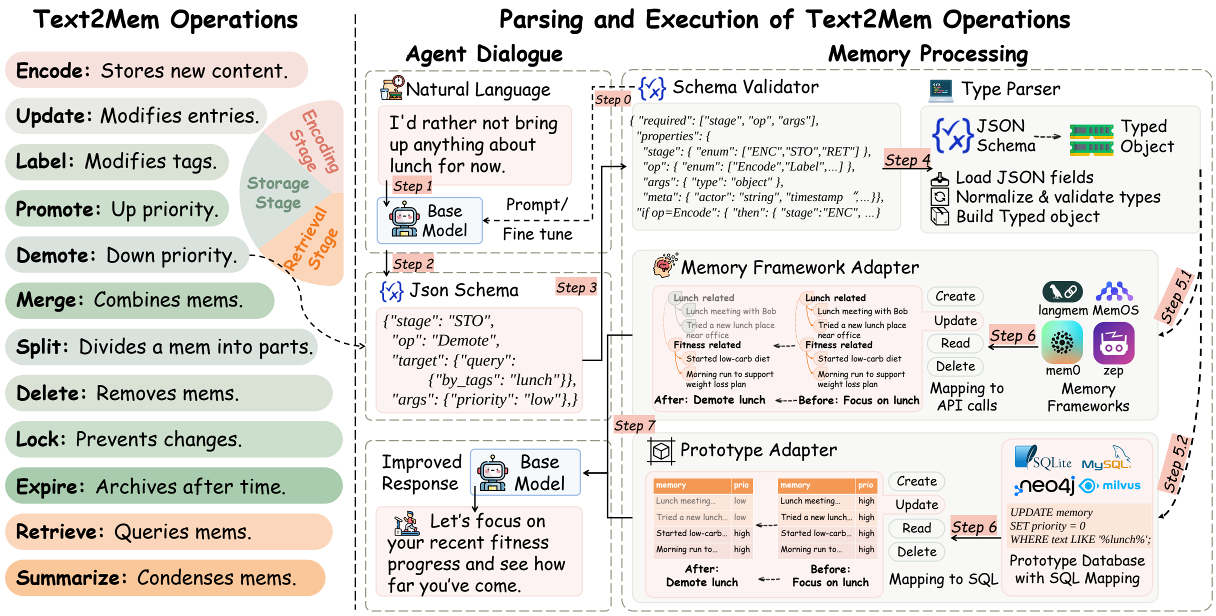

Figure 2: **Text2Mem execution pathway.** The base model (Step 0) learns Text2Mem schema rules via prompting or fine-tuning. A natural-language command (Step 1) is translated into a JSON schema (Step 2), validated (Step 3), and parsed into a typed operation object (Step 4). The object is then mapped to either real memory frameworks (Step 5.1) or a SQL-based prototype backend (Step 5.2), where the operation is executed (Step 6). The updated memory state is finally used by the model to generate improved, context-aware responses (Step 7).

et al., 2025).

Crucially, standardized benchmarks built on explicit schemas enabled systematic evaluation, from Spider and CoSQL to more recent efforts such as BIRD and LogicCat (Yu et al., 2018, 2019; Li et al., 2024; Liu et al., 2025a). These lessons motivate Text2Mem: memory commands today resemble early SQL, and require a compact operation language, schema-level constraints, and typed execution to achieve precise and reproducible behavior.

3 Text2Mem

To address the limitations of existing systems, we introduce **Text2Mem**, a unified memory operation language for translating natural-language instructions into executable memory operations. As illustrated in Figure 2, Text2Mem adopts a three-stage pipeline: natural-language utterances are mapped to schema-based memory operations with explicit fields and constraints; validated schemas are parsed into operation objects with normalized parameters; and these objects are executed via adapters on a SQL-based reference backend or existing memory frameworks.

Text2Mem defines the structure and scope of memory operations, but does not prescribe how individual operations are internally realized by a specific backend, allowing different systems to implement the same operation specification in their own ways. This design provides a structured pathway from natural-language instructions to concrete memory operations across different backends.

The remainder of this section describes the core components of Text2Mem, including its verb-centered operation set (Section 3.1), schema specification (Section 3.2), and the validator–parser–adapter pipeline (Section 3.3).

3.1 Operation Set Design

Text2Mem adopts a verb-centered operation inventory for memory control. The operation set covers the full memory lifecycle from encoding to retrieval within a single vocabulary, while avoiding redundant or overlapping verbs. Each operation is defined with a clear scope, allowing multiple operations to be composed when needed.

3.1.1 Memory Operation Set

The final inventory contains twelve verbs: one for *encoding*, nine for *storage*, and two for *retrieval* (Table 1). Together, they cover the entire lifecycle of agent memory without semantic overlap.

Encoding. Text2Mem treats encoding as semantic interpretation before storage. The **Encode** op-

Stage	Operation	Description	MemOS	mem0	Letta
Encoding	`Encode`	Insert a new memory with metadata; embeddings are optional and deferrable later.	✓	✓	✓
Storage	`Update`	Modify specific fields with validation, lineage safety, and strict type checks.	✓	△	△
	`Label`	Add, replace, or remove tags and edit facets with deduplication constraints.	△	△	△
	`Promote`	Increase priority or attach reminders to resurface items on a defined cadence.	–	–	–
	`Demote`	Decrease priority or archive without deleting data to reduce retrieval prominence.	–	–	–
	`Merge`	Combine records into a primary while preserving lineage links and optional child deletes.	–	–	–
	`Delete`	Soft or hard delete with policy and lock checks, including time-range filters.	✓	✓	✓
	`Split`	Break composite entries into smaller linked units via sentences or chunks.	–	–	–
	`Lock`	Restrict edits via read-only or append-only policies with reason and expiry metadata.	–	–	–
	`Expire`	Apply TTL or until, triggering actions when expired such as demote or anonymize.	–	–	–
Retrieval	`Retrieve`	Run filtered, ranked queries with permission-aware results and field-level whitelists.	✓	✓	✓
	`Summarize`	Produce concise, focused summaries within token limits aligned to a specified focus.	△	△	△

Table 1: The Text2Mem operation inventory and its support in existing frameworks. ✓: native support; △: partial or indirect; –: unsupported. Basic operations like `Encode`, `Delete`, and `Retrieve` are common, while higher-order controls remain absent.

eration resolves entities and time, attaches provenance, and assigns governance attributes prior to indexing, ensuring that every stored item is interpretable, governable, and portable across backends. Detailed encoding semantics and an illustrative algorithm are provided in Appendix A.

Storage. Storage operations govern how memories evolve after creation. Text2Mem elevates storage from CRUD-style data maintenance to semantic governance by introducing first-class controls for rewriting, prioritization, consolidation, and lifecycle management. In particular, operations such as **Promote** and **Demote** regulate memory salience without mutating content, while **Merge**, **Split**, **Lock**, and **Expire** enable consolidation, decomposition, safety, and temporal control. Detailed semantics of individual storage verbs are deferred to Appendix A.

Retrieval. Retrieval operations provide a controlled interface for bringing memory back into focus. **Retrieve** supports hybrid symbolic and semantic queries under governance constraints, and **Summarize** produces concise semantic digests that preserve what matters while keeping context compact. Further details are described in Appendix A.

3.1.2 Implications and Comparison

Text2Mem is not merely a collection of memory verbs, but a unified operation interface in which each verb has a clear scope and execution role. The verbs are composable, allowing agents to express multi-step memory workflows (e.g., `Retrieve` $\rightarrow$ `Label` $\rightarrow$ `Promote` $\rightarrow$ `Summarize`) using a consistent structure, without relying on ad-hoc backend extensions.

Why memory is not just a database. Although agent memory is often built atop persistent storage, Text2Mem does not treat memory as a conventional database abstraction. At a low level, agent memory can be viewed as a personalized data store coupled with a large language model; however, semantic interpretation and long-horizon reasoning impose requirements beyond CRUD-style operations. Memory items must be interpreted, consolidated, prioritized, and governed based on linguistic intent, motivating an explicit memory operation language that mediates between natural language and execution.

Table 1 compares Text2Mem with representative frameworks (MemOS (Li et al., 2025), mem0 (Chhikara et al., 2025), and Letta (Packer et al., 2024)). While basic operations such as **Encode**, **Delete**, and **Retrieve** are commonly supported, many higher-level operations required for memory governance—including **Promote**, **Demote**, **Merge**, **Split**, **Lock**, and **Expire**—remain absent or only partially implemented. This gap highlights the need for a standardized and expressive operation set for memory control, which Text2Mem aims to address through a unified specification independent of any single backend.

3.2 Operation Schema

A core difficulty in memory control is that natural-language instructions are often underspecified with respect to scope, timing, and permissions. Text2Mem introduces a compact **operation schema** to explicitly specify execution-relevant information prior to runtime. Each instruction is represented as a typed JSON object with required fields and cross-field constraints, which can be checked before execution across different backends.

The schema consists of five fields: **stage**, **op**, **target**, **args**, and **meta**. Together, these fields describe the operation type, affected scope, and associated parameters. Potentially destructive actions require explicit confirmation or dry-run flags, and lifecycle rules restrict updates to locked or expired items, filtering out invalid operations before execution.

Schema Architecture and Execution Semantics Each validated Text2Mem schema specifies a concrete memory-state transition. The **op** field selects a predefined verb, while **target** specifies the affected items using exactly one of four modes: explicit identifiers (**ids**), structured predicates (**filter**), semantic search (**search**), or global scope (**all**). For non-trivial scopes, explicit limits or confirmations are required to constrain the affected set prior to execution.

Operation-specific parameters are provided through **args** and normalized during validation. Cross-field invariants check governance constraints, such as preventing deletion of locked items or updates to expired ones, and reject invalid state transitions before execution. Execution applies the validated schema to the current memory state and records the resulting changes to affected items and governance attributes. Additional examples are provided in Appendix B.

3.3 Validator–Parser–Adapter Pipeline

Text2Mem processes each validated schema through a three-stage pipeline. The **validator** checks structural completeness and safety constraints; the **parser** converts the schema into a typed operation with normalized parameters; and the **adapter** applies the operation to either a SQL reference backend or an existing memory framework. Execution produces a structured **Execution-Result** that records the affected memory items and corresponding state changes.

3.3.1 Validator

The validator performs initial checks on schema instances. It verifies structural requirements and cross-field invariants, such as field completeness and basic safety constraints, and rejects schemas that violate these conditions. This stage shifts a portion of safety checks from runtime to validation, so that subsequent stages operate on well-formed schema instances.

3.3.2 Parser

The parser converts validated schemas into typed operation objects used internally by Text2Mem. During this process, parameters are normalized into explicit representations, such as concrete time ranges and priority values. This representation provides a uniform internal format for subsequent execution and inspection.

3.3.3 Adapter

The adapter maps typed operation objects to executable actions in different backends. It operates on schemas that have already been validated and normalized by earlier stages, and applies the corresponding operations using backend-specific interfaces. This design allows the same operation specification to be applied across different execution environments.

Design perspective. Text2Mem focuses on defining a common specification for memory operations, rather than providing a full integration layer for existing memory frameworks. The adapter layer is intended to illustrate how the specification can be applied to different systems, rather than to optimize any particular backend.

Experimental focus. Our evaluation uses a SQL-based reference adapter, which provides a controlled execution environment for inspecting memory state changes. The SQL prototype serves as a reference implementation for benchmarking and verification through declarative database assertions. Adapters to real-world frameworks follow the same operation specification and are discussed qualitatively in Appendix C. Large-scale empirical comparisons across multiple backends are left for future work.

4 Text2Mem Benchmark

Text2Mem Benchmark provides an end-to-end evaluation framework for assessing memory-centric reasoning systems. It evaluates not only how models translate natural-language instructions into formal memory operations, but also how faithfully these operations execute within a dynamic memory environment. By coupling language understanding with database-verifiable effects, the benchmark bridges intent interpretation and operational reliability within a unified and auditable framework.

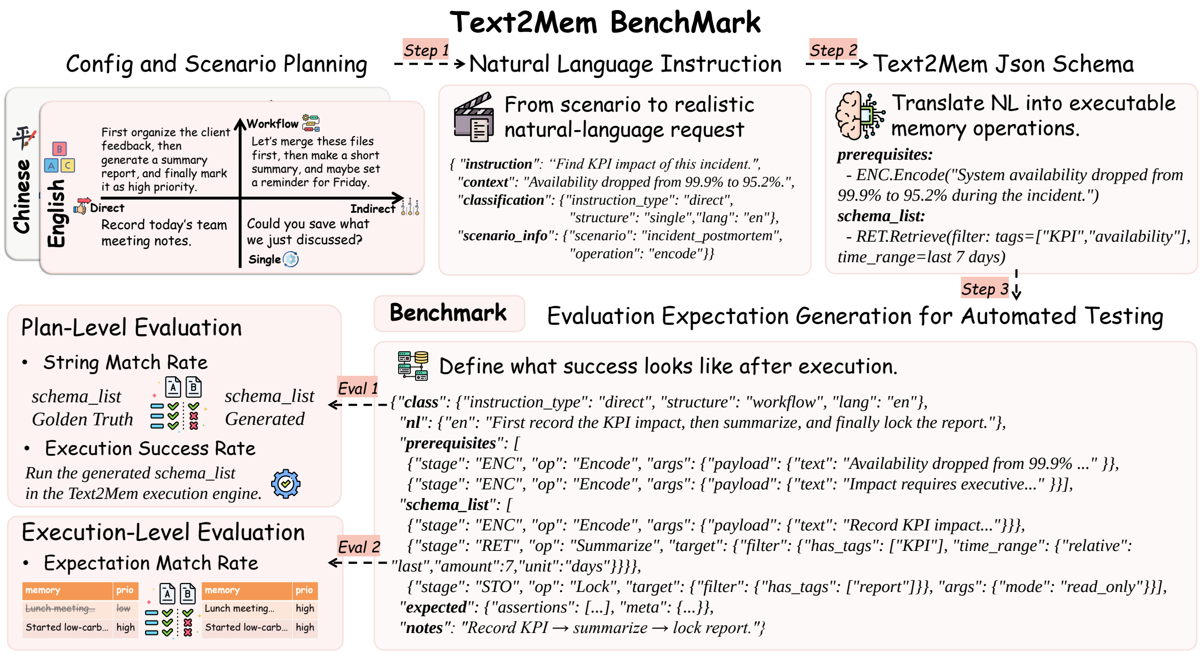

Figure 3: **Overview of the Text2Mem Benchmark pipeline.** The process consists of three generation stages and two evaluation layers. **Step 1:** Scenarios are converted into realistic natural-language requests. **Step 2:** The requests are translated into executable memory operation schemas (`schema_list`) with corresponding `prerequisites`. **Step 3:** Expected outcomes are defined for automatic verification after execution. Two evaluation layers assess system performance: **Plan-level evaluation** measures string-match accuracy and execution success rate of generated schemas; **Execution-level evaluation** measures expectation match rate and retrieval-based metrics to quantify behavioral correctness after running the `schema_list` in the Text2Mem system.

4.1 Evaluation Methodology

Text2Mem Benchmark evaluates two complementary layers: the *planning layer*, which measures how accurately a system translates natural-language instructions into valid Text2Mem schemas, and the *execution layer*, which assesses how these schemas produce verifiable effects in a memory environment. All experiments are conducted within a structured SQL prototype that mirrors real memory frameworks while remaining fully auditable.

Structured memory environment. We construct a unified memory database that captures the full semantics of Text2Mem operations. Each record is a typed and addressable item with fields for content, semantics, governance, and access control. The benchmark uses the following prototype schema:

```
CREATE TABLE IF NOT EXISTS memory (
    id INTEGER PRIMARY KEY
       AUTOINCREMENT,
    -- Content and semantics
    text TEXT,
    type TEXT,
    tags TEXT,          -- JSON array
facets TEXT,       -- JSON object {
    subject,time,location,topic},
weight REAL,
embedding TEXT,    -- JSON array
embedding_model TEXT,
-- Governance and lifecycle
source TEXT,
expire_at TEXT,
lock_mode TEXT,
lineage_parents TEXT,
lineage_children TEXT,
-- Access control
read_perm_level TEXT,
write_perm_level TEXT );
```

This prototype is used as a controlled execution environment for evaluating memory state changes, rather than as a model of any specific production memory system.

Planning (Natural Language → Text2Mem Schema). Given a natural-language instruction, the system must generate a memory operation schema that is structurally valid and well-formed with respect to the Text2Mem specification. For composite requests, the system produces an ordered sequence of operations (*schema_list*) that preserves execution dependencies across stages.

Execution (Text2Mem Schema → Real Effects). Given a validated schema (or schema list), the system must realize the intended operational effects in execution. The benchmark evaluates consistency across both the SQL-based reference backend and real framework adapters. All effects are verified through declarative database assertions, ensuring that formally correct schemas also yield functionally correct behavior.

Metrics and reproducibility. The benchmark reports metrics independently for the planning and execution layers.

At the planning layer, we measure **String Match Accuracy (SMA)** and **Execution Success Rate (ESR)**:

$$\mathrm{SMA} = \frac{1}{|\mathcal{S}|}\sum_{s\in\mathcal{S}} \mathrm{sim}(s, \mathcal{S}^*), \quad \mathrm{ESR} = \frac{1}{|\mathcal{S}|}\sum_{s\in\mathcal{S}} \psi(s),$$

where $\mathrm{sim}(\cdot,\cdot)$ combines normalized Levenshtein distance and cosine similarity, and $\psi(\cdot)$ indicates successful execution.

At the execution layer, performance is quantified by the **Expectation Match Rate (EMR)**, defined as

$$\mathrm{EMR} = \frac{\sum_i \sum_{a\in\mathcal{A}_i} \mathbf{1}[a]}{\sum_i |\mathcal{A}_i|}.$$

EMR measures the proportion of satisfied SQL-level assertions, capturing whether executed schemas produce the expected memory state changes. Together, SMA and ESR characterize schema-level structure and executability, while EMR reflects execution-level memory effects.

4.2 Dataset Construction

Text2Mem Benchmark instances are constructed to balance coverage, realism, and reproducibility. We organize the dataset along four dimensions: instruction type (direct vs. indirect), operational structure (single vs. workflow), language, and evaluation layer. This design ensures systematic coverage of realistic memory-use scenarios while keeping the benchmark compact and interpretable.

Each instance is grounded in a semantically coherent context synthesized from realistic work artifacts, and is automatically validated under the Text2Mem schema. All instances are linked to executable SQL assertions, enabling database-verifiable transitions from instruction to effect.

Detailed dataset taxonomy, generation procedures, and configuration statistics are provided in Appendix E.

4.3 Experiment Design and Results

To evaluate the behavior of models under the Text2Mem Benchmark, we conduct controlled experiments across both the planning and execution layers. The goal is to assess how well large language models translate natural-language memory instructions into well-formed operation schemas, and how these schemas behave when executed in a controlled memory environment.

Experimental setup. We evaluate a set of representative foundation models on their ability to generate valid Text2Mem schemas. All experiments are conducted within a SQLite-based prototype environment, which provides a controlled setting for inspecting execution outcomes and memory state changes. We benchmark five models covering both open and proprietary systems: *GPT-4o*, *Qwen2.5-72B-Instruct*, *Claude-4-Sonnet*, *Gemini-2.5-Pro*, and *GPT-5*. Each model is given identical natural-language instructions under the same prompt template and produces candidate schemas. These schemas are then validated and executed within the benchmark pipeline to measure schema validity and execution-level behavior.

Metrics. Following Section 4, we report results using three complementary metrics. **String Match Accuracy** (**SMA**$_{\mathrm{lev}}$, **SMA**$_{\mathrm{cos}}$) measures structural alignment between generated schemas and reference schemas, using normalized Levenshtein distance and cosine similarity. **Execution Success Rate (ESR)** captures whether a generated schema can be successfully validated and executed without runtime errors. **Expectation Match Rate (EMR)** evaluates whether executing the generated operations produces the intended memory state changes, as verified by SQL-level assertions. Together, these metrics distinguish between schema-level correctness and execution-level behavioral fidelity.

Results. Tables 2 and 3 summarize model performance on the Text2Mem Benchmark. At the operation level (Table 2), models achieve consistently high Execution Success Rates (ESR), typically exceeding 90% across most operations. This indicates that the schema format and basic validation rules are generally easy for models to learn, and that generated schemas are usually executable within the benchmark environment. String Match Accuracy (SMA) is also relatively high across models, suggesting that models can align well with the structural form of reference schemas.

Metrics	Models	Encoding	Storage									Retrieval		Avg
		Encode	Update	Label	Promote	Demote	Merge	Split	Lock	Expire	Delete	Retrieve	Summarize	
$\mathrm{SMA}_{\mathrm{lev}}$	GPT-4o	0.5466	0.6704	0.7468	0.6900	0.7161	0.6829	0.7385	0.6404	0.7549	0.6929	0.6936	0.7335	0.6776
	Qwen2.5-72B-I	0.5530	0.6490	0.7058	0.6957	0.7710	0.5423	0.7792	0.6264	0.7437	0.6759	0.6571	0.6789	0.6195
	Claude-4-Sonnet	0.5501	0.5953	0.6309	0.6274	0.7026	0.4737	0.5508	0.6074	0.8268	0.6562	0.5978	0.6586	0.6230
	Gemini-2.5-Pro	0.5421	0.6642	0.6975	0.6835	0.7227	0.5716	0.6771	0.5760	0.8003	0.6440	0.6630	0.6786	0.6491
	GPT-5	0.5185	0.5624	0.6585	0.6183	0.6828	0.6225	0.4595	0.5495	0.7540	0.6267	0.5914	0.5447	0.6115
$\mathrm{SMA}_{\mathrm{cos}}$	GPT-4o	0.8005	0.8845	0.8813	0.8834	0.9111	0.8590	0.8613	0.8647	0.9088	0.9233	0.8504	0.8797	0.8690
	Qwen2.5-72B-I	0.8021	0.8430	0.8703	0.9173	0.9595	0.7690	0.9152	0.8757	0.9078	0.9344	0.8483	0.8661	0.8349
	Claude-4-Sonnet	0.8123	0.8123	0.8302	0.8502	0.9097	0.7653	0.7602	0.8747	0.9571	0.8895	0.8022	0.8705	0.8475
	Gemini-2.5-Pro	0.8026	0.8684	0.8720	0.8801	0.9152	0.7865	0.8500	0.8461	0.9405	0.8864	0.8492	0.8662	0.8560
	GPT-5	0.7911	0.8328	0.8521	0.8638	0.9002	0.8505	0.7180	0.8206	0.9316	0.8939	0.8175	0.8164	0.8396
ESR	GPT-4o	0.9853	0.9178	0.9559	0.9326	0.9627	0.9884	0.9633	0.9453	0.7500	0.9328	0.9910	0.9758	0.9484
	Qwen2.5-72B-I	1.0000	0.9583	0.9286	0.7330	0.9695	0.9123	0.2353	0.9535	0.7143	0.9647	0.9874	0.9691	0.9510
	Claude-4-Sonnet	1.0000	0.9273	0.9211	0.9744	0.8095	0.9806	0.3571	0.9444	0.6923	0.8636	0.9911	0.9844	0.9594
	Gemini-2.5-Pro	0.9953	0.9762	0.9897	0.9574	0.8936	1.0000	0.1000	1.0000	0.7273	0.9412	0.9796	1.0000	0.9479
	GPT-5	1.0000	1.0000	0.9639	0.9800	0.9259	1.0000	0.3750	1.0000	0.8182	1.0000	0.9873	1.0000	0.9662
EMR	GPT-4o	0.3462	0.0514	0.1735	0.1140	0.2547	0.0465	0.0000	0.0859	0.0294	0.1176	0.0328	0.3945	0.2532
	Qwen2.5-72B-I	0.4255	0.0833	0.1071	0.2521	0.0976	0.0000	0.0000	0.1860	0.1571	0.2625	0.0938	0.2917	0.2535
	Claude-4-Sonnet	0.4640	0.0364	0.1316	0.1538	0.1429	0.0000	0.0000	0.3333	0.0769	0.1364	0.0625	0.4063	0.3132
	Gemini-2.5-Pro	0.3953	0.1071	0.2577	0.1702	0.3191	0.0000	0.0000	0.2727	0.0909	0.0588	0.0306	0.5766	0.2897
	GPT-5	0.3227	0.1429	0.2048	0.2200	0.2593	0.0000	0.0000	0.0588	0.1818	0.1667	0.1709	0.4818	0.2826

Table 2: Model performance on Text2Mem Benchmark rearranged by column order: Metrics, Models, Encoding, Storage, Retrieval, and Avg. Metrics include $\mathrm{SMA}_{\mathrm{lev}}$, $\mathrm{SMA}_{\mathrm{cos}}$, ESR, and EMR. Cells with light blue shading indicate the best-performing model for each metric.

Models	Metrics	Language		Instruction Type		Structure	
		Zh	En	Direct	Indirect	Single	Workflow
GPT-4o	$\mathrm{SMA}_{\mathrm{lev}}$	0.677	0.678	0.678	0.676	0.679	0.665
	$\mathrm{SMA}_{\mathrm{cos}}$	0.865	0.873	0.867	0.864	0.867	0.858
	ESR	0.917	0.980	0.928	0.926	0.927	0.933
	EMR	0.183	0.323	0.199	0.225	0.218	0.088
Qwen2.5-72B-I	$\mathrm{SMA}_{\mathrm{lev}}$	0.6706	0.5683	0.6630	0.6435	0.6546	0.6851
	$\mathrm{SMA}_{\mathrm{cos}}$	0.8634	0.8065	0.8598	0.8468	0.8528	0.8907
	ESR	0.9021	1.0000	0.9211	0.9020	0.9145	0.9231
	EMR	0.1888	0.3182	0.2018	0.2157	0.2171	0.0769
Claude-4 Sonnet	$\mathrm{SMA}_{\mathrm{lev}}$	0.6416	0.6044	0.6372	0.6321	0.6370	0.6203
	$\mathrm{SMA}_{\mathrm{cos}}$	0.8424	0.8525	0.8456	0.8434	0.8426	0.8599
	ESR	0.9343	0.9844	0.9593	0.9348	0.9425	0.9412
	EMR	0.2358	0.3906	0.2520	0.2645	0.2712	0.1471
Gemini-2.5-Pro	$\mathrm{SMA}_{\mathrm{lev}}$	0.6464	0.6519	0.6496	0.6480	0.6525	0.5961
	$\mathrm{SMA}_{\mathrm{cos}}$	0.8490	0.8630	0.8574	0.8525	0.8570	0.8411
	ESR	0.9239	0.9719	0.9490	0.9453	0.9462	0.9762
	EMR	0.2310	0.3483	0.2687	0.2980	0.2990	0.1429
GPT-5	$\mathrm{SMA}_{\mathrm{lev}}$	0.5983	0.6248	0.6140	0.6053	0.6144	0.5668
	$\mathrm{SMA}_{\mathrm{cos}}$	0.8291	0.8501	0.8393	0.8404	0.8406	0.8234
	ESR	0.9465	0.9860	0.9667	0.9652	0.9641	1.0000
	EMR	0.2197	0.3455	0.2587	0.2922	0.2930	0.1190

Table 3: Model performance on Text2Mem Benchmark by language, instruction type, and structure. Metrics include $\mathrm{SMA}_{\mathrm{lev}}$, $\mathrm{SMA}_{\mathrm{cos}}$, ESR, and EMR. Cells with light blue shading indicate the better-performing type within each comparison.

In contrast, Expectation Match Rate (EMR) is substantially lower. Although models often generate grammatically valid and executable schemas, the resulting memory state changes frequently deviate from the expected outcomes. This suggests that generating structurally valid schemas is substantially easier than reasoning about stateful memory effects, rather than reflecting a limitation of the schema design or its expressiveness. This gap is most pronounced for complex storage operations such as *Merge* and *Split*, which require compositional reasoning over multiple memory items.

Across models, *GPT-4o* achieves the highest structural similarity, *GPT-5* attains the highest execution success rate, and *Claude-4-Sonnet* records the highest expectation match rate. From a global perspective (Table 3), English instructions outperform Chinese ones, and single-operation tasks consistently outperform multi-step workflows, reflecting the additional difficulty introduced by compositional execution. Overall, these results indicate that while Text2Mem lowers the barrier to producing well-formed and executable schemas, accurately realizing the intended memory effects remains a key challenge. Additional diagnostic analyses are provided in Appendix F.

5 Conclusion

This paper introduced **Text2Mem**, a unified memory operation language for LLM-based agents that provides a structured way to translate natural-language instructions into explicit memory operations. Text2Mem defines a compact set of operations together with a schema-based specification that makes the structure, scope, and parameters of memory actions explicit, while leaving their realization to individual backends. We also presented the **Text2Mem Benchmark**, a two-layer evaluation suite that separates schema generation from execution-level effects, enabling systematic analysis of both planning behavior and stateful memory reasoning. Together, Text2Mem and its benchmark offer a common reference point for studying and comparing memory control in LLM-based agents, and highlight execution-level reasoning as a key challenge for future work.

Limitations

This work has several limitations. (1) Text2Mem focuses on language-level specification and schema-based execution, and does not provide formal semantic guarantees for complex memory behaviors beyond structural constraints and execution checks; higher-order memory reasoning therefore remains dependent on model behavior. (2) Our evaluation relies on a SQL-based reference backend and a limited set of existing memory frameworks, which enables controlled and auditable execution but does not capture the full diversity of real-world memory architectures. (3) The benchmark evaluates schema generation and memory state transitions rather than end-to-end agent performance, such as task success or long-horizon behavior, which we leave for future work.

References

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. 2025. Mem0: Building production-ready ai agents with scalable long-term memory. *Preprint*, arXiv:2504.19413.

Longxu Dou, Yan Gao, Mingyang Pan, Dingzirui Wang, Wanxiang Che, Dechen Zhan, and Jian-Guang Lou. 2022. Unisar: A unified structure-aware autoregressive language model for text-to-sql. *Preprint*, arXiv:2203.07781.

Bernal Jimenez Gutierrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. 2024. Hipporag: Neurobiologically inspired long-term memory for large language models. In *Advances in Neural Information Processing Systems 38*.

Bernal Jiménez Gutiérrez, Yiheng Shu, Weijian Qi, Sizhe Zhou, and Yu Su. 2025. From rag to memory: Non-parametric continual learning for large language models. *CoRR*, abs/2502.14802.

Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2025. Next-generation database interfaces: A survey of llm-based text-to-sql. *Preprint*, arXiv:2406.08426.

Jimin Lee, Ingeol Baek, Byeongjeong Kim, Hyunkyung Bae, and Hwanhee Lee. 2025. Safe-sql: Self-augmented in-context learning with fine-grained example selection for text-to-sql. *Preprint*, arXiv:2502.11438.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2024. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36.

Zhiyu Li, Shichao Song, Chenyang Xi, Hanyu Wang, Chen Tang, Simin Niu, Ding Chen, Jiawei Yang, Chunyu Li, Qingchen Yu, Jihao Zhao, Yezhaohui Wang, Peng Liu, Zehao Lin, Pengyuan Wang, Jiahao Huo, Tianyi Chen, Kai Chen, Kehang Li, and 20 others. 2025. Memos: A memory os for ai system. *arXiv preprint arXiv:2507.03724*.

Xiang Liang, Simin Niu, Zhiyu Li, Sensen Zhang, Shichao Song, Hanyu Wang, Jiawei Yang, Feiyu Xiong, Bo Tang, and Chenyang Xi. 2024. Empowering large language models to set up a knowledge retrieval indexer via self-learning. *CoRR*, abs/2405.16933.

Dominik Lindorfer. 2023. Sql-llama: Text-2-sql using an instruction-following llama-2 model. `https://github.com/dominiklindorfer/SQL-LLaMA`.

Tao Liu, Xutao Mao, Hongying Zan, Dixuan Zhang, Yifan Li, Haixin Liu, Lulu Kong, Jiaming Hou, Rui Li, YunLong Li, aoze zheng, Zhiqiang Zhang, Luo Zhewei, Kunli Zhang, and Min Peng. 2025a. Logiccat: A chain-of-thought text-to-sql benchmark for complex reasoning. *Preprint*, arXiv:2505.18744.

Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and Jingren Zhou. 2025b. Xiyan-sql: A novel multi-generator framework for text-to-sql. *Preprint*, arXiv:2507.04701.

Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, Rongcheng Tu, Xiao Luo, Wei Ju, Zhiping Xiao, Yifan Wang, Meng Xiao, Chenwu Liu, Jingyang Yuan, Shichang Zhang, and 7 others. 2025. Large language model agent: A survey on methodology, applications and challenges. *Preprint*, arXiv:2503.21460.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. Memgpt: Towards llms as operating systems. *Preprint*, arXiv:2310.08560.

Ge Qu, Jinyang Li, Bowen Li, Bowen Qin, Nan Huo, Chenhao Ma, and Reynold Cheng. 2024. Before generation, align it! a novel and effective strategy for mitigating hallucinations in text-to-SQL generation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 5456–5471, Bangkok, Thailand. Association for Computational Linguistics.

Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. 2025. Zep: A temporal knowledge graph architecture for agent memory. *CoRR*, abs/2501.13956.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578, Online. Association for Computational Linguistics.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6).

Jiale Wei, Xiang Ying, Tao Gao, Fangyi Bao, Felix Tao, and Jingbo Shang. 2025. Ai-native memory 2.0: Second me. *Preprint*, arXiv:2503.08102.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen LLM applications via multi-agent conversation framework. *CoRR*, abs/2308.08155.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2025a. A-mem: Agentic memory for llm agents. *Preprint*, arXiv:2502.12110.

Ziwen Xu, Shuxun Wang, Kewei Xu, and 1 others. 2025b. Easyedit2: An easy-to-use steering framework for editing large language models. *Preprint*, arXiv:2504.15133.

Hongkang Yang, Zehao Lin, Wenjin Wang, Hao Wu, Zhiyu Li, Bo Tang, Wenqiang Wei, Jinbo Wang, Zeyun Tang, Shichao Song, Chenyang Xi, Yu Yu, Kai Chen, Feiyu Xiong, Linpeng Tang, and Weinan E. 2024. Memory^3: Language modeling with explicit memory. *Journal of Machine Learning*, 3(3):300–346.

Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, and 5 others. 2019. Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1962–1979, Hong Kong, China. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Ningyu Zhang, Yunzhi Yao, Bozhong Tian, and 1 others. 2024. A comprehensive study of knowledge editing for large language models. *Preprint*, arXiv:2401.01286.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, and 3 others. 2025. A survey of large language models. *Preprint*, arXiv:2303.18223.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *Preprint*, arXiv:1709.00103.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 2024. Memorybank: enhancing large language models with long-term memory. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'24/IAAI'24/EAAI'24. AAAI Press.

A Detailed Semantics of Text2Mem Operations

This appendix provides detailed semantic explanations and illustrative algorithms for the Text2Mem operation set, complementing the high-level overview in Section 3.1.

A.1 Encoding Semantics

The encoding stage defines how new information becomes structured memory rather than raw data. In Text2Mem, each memory item is a typed and addressable object that integrates content, contextual attributes (who, when, where), semantic facets (tags, entities, topics), governance fields (priority, permissions, lifespan), and machine-usable signals such as embeddings and summaries.

The **Encode** operation performs semantic interpretation before storage: it resolves entities and time, attaches provenance, extracts tasks or decisions, and sets governance attributes that determine how the item should persist. Only after meaning and scope are clarified does it generate indices, embeddings, and concise summaries. This design ensures that every stored item carries explicit semantics and provenance, making it interpretable and portable across heterogeneous backends.

A.2 Storage Semantics

The storage stage governs how memories evolve after creation. Unlike traditional databases centered on CRUD operations over opaque rows, Text2Mem treats storage as semantic governance over well-formed memory items.

Update enables semantic rewriting rather than blind overwriting, allowing models to improve clarity, normalize entities and time, or expand terse

Algorithm 1 Encode Operation

Require: Raw content C, context metadata M
Ensure: Schema-compliant memory item $\mathcal{M}$

$\hat{C} \leftarrow \text{Resolve}(C, M)$
$\mathcal{E} \leftarrow \text{Extract}(\hat{C})$
$\mathcal{P} \leftarrow \text{Provenance}(M)$
$\mathcal{G} \leftarrow \text{Governance}(M)$
$\mathcal{R} \leftarrow \text{Represent}(\hat{C}, \mathcal{E})$
$\mathcal{M} \leftarrow \text{Assemble}(\mathcal{E}, \mathcal{P}, \mathcal{G}, \mathcal{R})$
return $\mathcal{M}$

notes into structured tasks. **Label** infers and aligns tags, entities, and topics with existing taxonomies. **Delete** supports governance-aware removal, including soft deletion with recovery windows and hard deletion subject to locks and audit trails.

Promote and **Demote** introduce priority as a first-class control for memory salience, influencing ranking, recency decay, and reminder cadence without mutating content. **Merge** and **Split** operate at semantic granularity: related items can be consolidated into a coherent entry with preserved lineage, while multi-topic notes can be decomposed into atomic pieces for finer control.

Lock and **Expire** provide safety and lifecycle guarantees. Locks enforce read-only or append-only modes for sensitive items, while expiration attaches time-to-live or concrete deadlines with explicit post-expiry behavior such as archival or demotion.

A.3 Retrieval Semantics

The retrieval stage brings memory back into focus for reasoning and reuse.

Retrieve issues filtered and ranked queries using hybrid symbolic and embedding-based strategies, respecting governance fields such as permissions, priority, soft deletes, and expiration. **Summarize** performs semantic condensation over retrieved items, producing concise narratives or structured digests that preserve decisions, action items, and open questions while keeping working context compact.

B Extended Schema Examples and Constraints

This appendix provides extended workflow examples and detailed constraint illustrations for the Text2Mem operation schema, complementing the high-level specification in Section 3.2.

B.1 Illustrative Multi-step Workflows

Beyond atomic operations, Text2Mem supports multi-step workflows where each stage is independently validated yet semantically coherent. The following examples demonstrate how the schema generalizes from targeted control to system-level governance.

B.1.1 Semantic Promotion Workflow

A project owner wants to raise the importance of all OKR(objective tracking notes)-related notes while leaving unrelated entries unchanged. The natural-language instruction is: *"Please prioritize everything about this quarter's OKR (objective tracking notes) progress and make sure those items stay visible in my task view."*

Step 1: Encode relevant and background notes.

```
{"stage":"ENC","op":"Encode",
 "args":{"payload":{"text":"Q2 OKR review
    meeting notes: marketing reach increased
    by 25\%, sales pipeline ahead of target,
    and product adoption remains steady."},
         "tags":["OKR","review","meeting"],
         "type":"note",
         "time":"2025-04-10T15:30:00+08:00",
         "source":"meeting_minutes",
         "facets":{"subject":"Q2 performance",
    "topic":"OKR progress"}}}

{"stage":"ENC","op":"Encode",
 "args":{"payload":{"text":"Action item:
    follow up with design team on the new
    dashboard for OKR tracking by next Monday.
    "},
         "tags":["OKR","task","dashboard"],
         "type":"task",
         "time":"2025-04-11T10:00:00+08:00",
         "source":"project_tracker",
         "facets":{"subject":"dashboard design
    ","topic":"task follow-up"}}}
```

Step 2: Promote via semantic search.

```
{"stage":"STO","op":"Promote",
 "target":{"search":{"intent":{"query":"OKR
    progress"},
                     "overrides":{"limit":5},
                     "limit":5}},
 "args":{"weight":0.9}}
```

B.1.2 Incident Postmortem Archive Workflow

After a SEV-1 network outage, an SRE (system reliability context) engineer documents the incident, locks the records for audit, and generates an executive summary.

Step 1: Encode incident timeline.

```
{"stage":"ENC","op":"Encode",
 "args":{"payload":{"text":"SEV-1 (high-
    severity incident) network incident
    timeline: 20:07 alert triggered by API
    latency, 20:12 automatic failover to
    backup nodes, 20:28 routing
    misconfiguration identified, 20:41 hotfix
    deployed, 20:48 service restored to 95\%
    availability."},
        "tags":["incident:p1-network","
    postmortem","owner:sre-ling"],
        "type":"incident_timeline",
        "time":"2025-09-28T22:30:00+08:00",
        "facets":{"subject":"2025-09-28 API
    Outage","topic":"incident_response"},
        "location":"cn-shanghai"}}
```

Step 2: Lock incident records.

```
{"stage":"STO","op":"Lock",
 "target":{"filter":{"has_tags":["incident:p1-
    network"],
                  "time_range":{"start":"
    2025-09-28T00:00:00+08:00",
                               "end":"
    2025-10-05T23:59:59+08:00"},
                  "limit":200}},
 "args":{"mode":"read_only",
        "reason":"Preserve SEV-1 incident
    records for compliance and leadership
    audit",
        "policy":{"allow":["Retrieve","
    Summarize"],
                 "deny":["Update","Delete"],
                 "reviewers":["
    oncall_manager","sre_lead"],
                 "expires":"2025-12-31T23
    :59:59+08:00"}},
 "meta":{"actor":"sre-ling","timestamp":"
    2025-09-29T00:05:00+08:00"}}
```

Step 3: Generate postmortem summary.

```
{"stage":"RET","op":"Summarize",
 "target":{"search":{"intent":{"query":"
    2025-09-28 API outage follow-up",
                                 "context":"
    executive briefing"},
                  "overrides":{"k":8,"
    order_by":"time_desc"},
                  "limit":8}},
 "args":{"focus":"Summarize root cause,
    customer impact, and assigned remediation
    owners."}}
```

B.1.3 Additional Examples in the Codebase

Beyond the illustrative examples included in the paper, the code repository provides a broader collection of executable examples that cover a wider range of operation patterns and usage scenarios. These examples are organized under the `examples/` directory and include:

- `ir_operations`, which demonstrates individual Text2Mem operations in isolation;
- `op_workflows`, which illustrates multi-step operation sequences and compositional memory workflows; and
- `real_world_scenarios`, which contains end-to-end examples derived from realistic application contexts.

All examples follow the same schema specification and execution pipeline described in the paper, and are intended to complement the abstract descriptions with concrete, runnable instances.

C Execution Pipeline Details

This appendix provides implementation-oriented details of the Validator–Parser–Adapter pipeline, complementing the high-level execution model described in Section 3.3.

C.1 Validator Rules

The validator enforces both structural and semantic correctness of Text2Mem schema instances. At the structural level, it verifies required fields, data types, and enumerated values. At the semantic level, it checks cross-field invariants such as prohibiting hard deletion of locked items, requiring finite horizons for expiration, and enforcing explicit confirmation for global write operations. Violations result in structured diagnostics identifying the failing field and rule.

C.2 Parser Normalization

The parser converts validated schemas into typed operation objects, the canonical internal representation of Text2Mem. All parameters are normalized into explicit and machine-determinable forms, including time ranges, priorities, and access rules. Implicit relations are expanded into explicit key–value pairs, and inconsistencies or missing references cause parsing to halt with diagnostic errors.

C.3 Adapter Mapping

Mapping to real memory frameworks. For operational memory systems such as MemGPT, mem0, or Letta, the adapter translates typed operation objects into framework-specific API sequences at the semantic level. For example, a **Promote** operation may be decomposed into priority adjustment and scheduled notification, while a **Lock** operation configures access-control policies. **Merge** and

Summarize operations may invoke retrieval and model-assisted consolidation when native support is unavailable.

Mapping to the SQL reference backend. Text2Mem also provides a SQL-based prototype backend that compiles operations into relational queries augmented with optional language-model calls. Simple operations map directly to `INSERT`, `UPDATE`, or `SELECT` statements, while complex verbs such as **Merge** or **Summarize** are executed as hybrid symbolic–semantic workflows. This backend serves as a transparent and auditable reference implementation for benchmarking and verification.

C.4 Model Integration

Certain operations invoke language-model services, such as embedding generation during **Encode** or abstraction during **Summarize**. Derived outputs are reattached to the memory backend through the same unified interface, preserving schema-level traceability.

D Benchmark Implementation Details

This appendix provides implementation-level details of the Text2Mem Benchmark, complementing the evaluation methodology described in Section 4.

D.1 SQL Prototype Schema Rationale

The SQL prototype schema captures the essential dimensions of Text2Mem memory items, including content (text, type), semantics (tags, facets, embeddings), governance (source, locks, lineage), and access control. Additional fields for logging, auditing, and provenance tracking are implemented in the full benchmark but omitted from the main paper for clarity.

D.2 Scenario Construction

Each benchmark instance begins with a blank memory database to ensure a clean and reproducible environment. A sequence of predefined setup operations inserts notes, tasks, events, and references to construct a semantically coherent context. This procedurally generated context serves as the interpretive background for subsequent instructions, enabling controlled retrieval, modification, and governance of memory items.

D.3 Execution-Level Effects

At the execution layer, schema instances are evaluated by verifying their concrete effects on the memory state. These effects include state transitions for editing operations, changes in priority or salience, lineage updates for consolidation operations, and temporal triggers such as reminders or expirations. All effects are verified through declarative SQL assertions, ensuring database-level observability and reproducibility.

E Dataset Construction Details

This appendix provides detailed descriptions of dataset construction for the Text2Mem Benchmark, complementing the high-level overview in Section 4.

E.1 Scenario Planning

Benchmark instances are organized along a four-way taxonomy that controls linguistic difficulty, operational structure, language, and evaluation layer. Each scenario defines a minimal yet semantically rich context derived from realistic sources such as meeting notes, task trackers, and project documentation, ensuring that memory operations are evaluated under plausible work and knowledge-management situations.

Instruction type: direct vs. indirect. *Direct* instructions explicitly specify both the intended operation and its parameters, primarily testing structural accuracy and schema formatting. *Indirect* instructions encode intent implicitly through pragmatic cues or conversational context, requiring models to infer missing arguments and resolve underspecified language into executable memory operations.

Structure: single vs. workflow. *Single* instructions correspond to atomic operations evaluated independently. *Workflow* instructions compose multiple dependent operations that share intermediate state, typically three to five steps, and are executed transactionally to verify inter-step consistency and final outcomes.

Language coverage. Each instance is provided in English (nl.en) or Chinese (nl.zh), with a subset offering bilingual pairs to support cross-lingual analysis of schema grounding and planning fidelity.

Practical configuration. The dataset includes four primary scenarios: *Incident Postmortem*, *Meeting Notes*, *Project Tracking*, and *Knowledge Base*. In the current release, approximately 85% of instructions are direct and 15% are indirect; 90%

Stage	Op	Pre-state	Post-state	Check expression
Encoding	Encode	ID not in DB	New record with content and metadata	Δcount = +1
Storage	Update	Record exists	Fields updated; lineage preserved	val_aft=exp $\wedge$ lid_aft=lid_bef
	Label	Tags/facets exist	Tag set modified, deduped	DISTINCT(tags) $\wedge$ tags_aft $\neq$ tags_bef
	Promote	weight=w_0	weight increased or trigger added	weight_aft $>$ weight_bef $\vee$ EXISTS(trigger)
	Demote	weight=w_0	weight decreased; record active	weight_aft $<$ weight_bef
	Merge	Multi-ID src	Children merged; lineage linked	merged_into $\neq$ NULL $\wedge$ count(child)=1
	Delete	Record active	Flagged deleted or removed	Δcount = -n
	Split	Composite record	Child records linked to parent	count(child)>1 $\wedge$ pid=src
	Lock	Editable record	Lock set (RO/AO)	lock $\in$ {RO,AO}
	Expire	TTL unset	Expiry registered; trigger active	expiry $\neq$ NULL $\wedge$ EXISTS(trigger)
Retrieval	Retrieve	Query-matching records	Results ranked and filtered	ids=exp $\wedge$ rank=exp
	Summarize	Context records	Summary stored with refs	sim $\geq \tau$

Table 4: Structured expectation templates for the twelve Text2Mem operations. Each operation is defined by its pre-state, post-state, and verification expression, enabling automated evaluation through SQL assertions.

correspond to single operations and 10% to multi-step workflows; all released samples are in English. This configuration ensures balanced coverage across practical memory contexts while maintaining reproducibility.

E.2 Dataset Generation Process

The construction of Text2Mem Benchmark follows a three-stage pipeline that mirrors the planning–execution workflow.

Stage I: Context synthesis. Raw materials are collected from realistic work and knowledge traces such as meeting minutes, task logs, and collaborative notes. A minimal but semantically rich context is synthesized for each test case, containing heterogeneous items (notes, tasks, events, and references). This contextual grounding provides the environmental state against which subsequent instructions are interpreted.

Stage II: Schema generation. Natural-language instructions—either direct or indirect, single or workflow—are paired with corresponding Text2Mem schemas through a semi-automatic annotation pipeline. Automatic templates are first generated using high-precision schema synthesis rules and model-assisted alignment, followed by human verification. Each finalized schema conforms to the official JSON specification and is independently validated through structural parsing.

Stage III: Assertion binding. For every schema, a set of declarative SQL assertions is automatically instantiated based on the operation type. These assertions follow the structured templates in Table 4, which define each Text2Mem operation by its expected pre-state, post-state, and verification expression. For instance, *Encode* checks record creation, *Promote* verifies increased weight or active triggers, and *Merge* or *Retrieve* confirm lineage linkage and result consistency. By translating each operation into explicit state–transition checks, the benchmark converts qualitative execution outcomes into quantitative, database-verifiable evidence, ensuring reproducibility across both layers.

Due to the length and complexity of the prompt templates used for both English and Chinese data synthesis, we provide the full generation prompts in the code repository under `bench/generate/prompts`.

Final Verification and Benchmark Compilation. Once all schemas and assertions have been validated for correctness, the fully verified instances are retained as the official benchmark dataset. This curated set ensures that all test cases are reproducible, accurate, and ready for consistent evaluation across models and frameworks. The benchmark is then packaged for public release, providing a robust and executable set of examples for future research and development.

F Extended Analysis of Benchmark Results

This appendix provides extended analysis and interpretation of the benchmark results reported in Section 4, focusing on diagnostic insights that go beyond headline metrics.

Syntactic validity versus semantic correctness. Across models, consistently high String Match Accuracy (SMA) and Execution Success Rate (ESR) indicate that the Text2Mem schema is easy for large language models to internalize and execute. This behavior reflects a deliberate design choice: by lowering the barrier of grammatical correctness and executability, Text2Mem minimizes confounding errors caused by formatting or schema misuse.

In contrast, the uniformly lower Expectation Match Rate (EMR) highlights a persistent gap between syntactic validity and faithful semantic execution. Although models often generate runnable JSON schemas, they frequently fail to satisfy SQL-level assertions that encode intent, governance constraints, and stateful effects. This gap suggests that higher-order reasoning over memory state transitions, rather than schema generation itself, remains the primary challenge.

Operation-level difficulty. Complex storage operations such as *Merge* and *Split* exhibit notably lower EMR scores. These operations require compositional reasoning over multiple memory items and their relationships, indicating that multi-step or state-dependent transformations are harder to realize accurately without targeted training or structured feedback.

Global trends and task complexity. At the aggregate level, English instructions outperform Chinese ones, and single-operation tasks achieve higher success rates than workflow-based compositions, consistent with the increased uncertainty introduced by compositional reasoning. Interestingly, although *direct* instructions yield higher SMA and ESR, they show lower EMR than *indirect* instructions. A plausible explanation is task-complexity confounding: explicitly phrased requests tend to encode harder, multi-constraint objectives, increasing the burden of SQL-level verification, whereas successfully parsed indirect instructions are often simpler, resulting in a survivorship-bias effect.