MemTensor

Text2Mem: A Unified Memory Operation Language for Memory Operating System

Felix Wang[1,2], Boyu Chen[1,2], Kerun Xu[1,2], Bo Tang[1,2], Feiyu Xiong[1,2], Zhiyu Li[1,2,†]

[1]MemTensor (Shanghai) Technology Co., Ltd., [2]Institute for Advanced Algorithms Research, Shanghai

Abstract

Large language model agents increasingly depend on memory to sustain long horizon interaction, but existing frameworks remain limited. Most expose only a few basic primitives such as encode, retrieve, and delete, while higher order operations like merge, promote, demote, split, lock, and expire are missing or inconsistently supported. Moreover, there is no formal and executable specification for memory commands, leaving scope and lifecycle rules implicit and causing unpredictable behavior across systems. We introduce Text2Mem, a unified memory operation language that provides a standardized pathway from natural language to reliable execution. Text2Mem defines a compact yet expressive operation set aligned with encoding, storage, and retrieval. Each instruction is represented as a JSON based schema instance with required fields and semantic invariants, which a parser transforms into typed operation objects with normalized parameters. A validator ensures correctness before execution, while adapters map typed objects either to a SQL prototype backend or to real memory frameworks. Model based services such as embeddings or summarization are integrated when required. All results are returned through a unified execution contract. This design ensures safety, determinism, and portability across heterogeneous backends. We also outline Text2Mem Bench, a planned benchmark that separates schema generation from backend execution to enable systematic evaluation. Together, these components establish the first standardized foundation for memory control in agents.

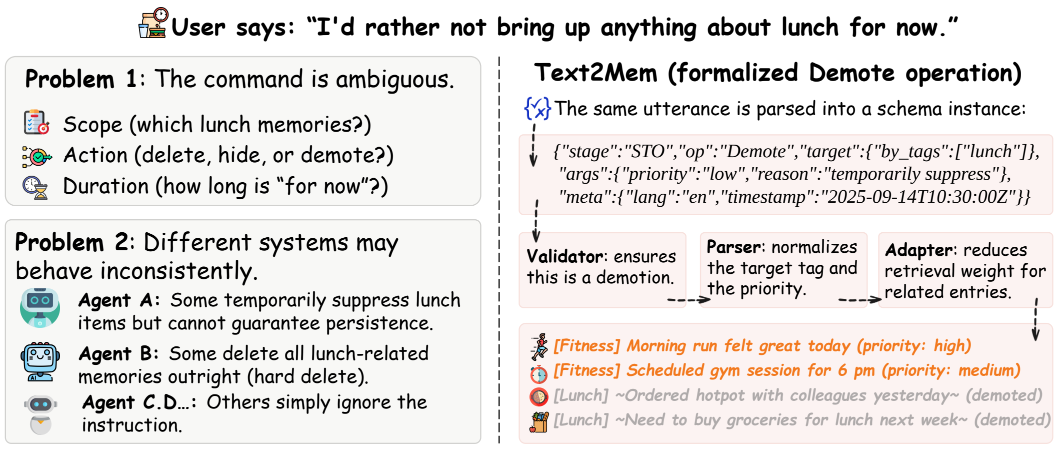
Author Legend: †Correspondence

Code: https://github.com/MemTensor/Text2Mem

1 Introduction

Large language model (LLM) agents[1–3] 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, which together enable persistent reasoning and personalized behavior [4–6].

Yet current memory subsystems remain rudimentary. Most frameworks expose only a small set of basic primitives such as encode, retrieve, and delete. Higher-order controls that are crucial for realistic use, including merge, promote or demote, split, lock, and expire, are either absent or only inconsistently implemented through ad-hoc extensions [7, 8]. This incompleteness creates several obstacles in practice. Portability across


User says: "I'd rather not bring up anything about lunch for now."
Problem 1: The command is ambiguous.
Scope (which lunch memories?)
Action (delete, hide, or demote?)
Duration (how long is "for now"?)
Problem 2: Different systems may behave inconsistently.
Agent A: Some temporarily suppress lunch items but cannot guarantee persistence.
Agent B: Some delete all lunch-related memories outright (hard delete).
Agent C.D...: Others simply ignore the instruction.
Text2Mem (formalized Demote operation)
The same utterance is parsed into a schema instance:
Validator: ensures this is a demotion.
Parser: normalizes the target tag and the priority.
Adapter: reduces retrieval weight for related entries.

```
{"stage":"STO","op":"Demote","target":{"by_tags":["lunch"]},
"args":{"priority":"low","reason":"temporarily suppress"},
"meta":{"lang":"en","timestamp":"2025-09-14T10:30:00Z"}}
```

[Fitness] Morning run felt great today (priority: high)
[Fitness] Scheduled gym session for 6 pm (priority: medium)
[Lunch] ~~Ordered hotpot with colleagues yesterday~~ (demoted)
[Lunch] ~~Need to buy groceries for lunch next week~~ (demoted)

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. Scope, action, and duration are unclear, leading to inconsistent behaviors across agents (temporary suppression, hard deletion, or ignoring). **Right**: Text2Mem resolves the ambiguity by instantiating a schema-based `Demote` operation with explicit arguments. The validator, parser, and adapter guarantee consistent execution across heterogeneous backends.

systems is limited because the same intent must be redefined for each framework. Compositional task design is difficult because some verbs overlap while others are missing entirely. More fundamentally, everyday use is constrained when users cannot express essential operations for organizing, protecting, or managing the lifecycle of their memories.

A second obstacle is the lack of a formal and executable specification for memory operations. Natural language commands such as "get rid of old notes" or "make rent top priority" are inherently underspecified: the scope, the mode of deletion, and the governance rules remain implicit. Without a schema that enforces required fields and invariants, and without typed objects that normalize values such as time ranges and priorities, systems cannot determine execution reliably. From the user's perspective, this means memory commands may behave unpredictably, vary across platforms, or even fail silently. From the developer's perspective, the absence of a shared specification makes it impossible to design consistent behaviors or to ensure that extensions will interoperate safely.

As an illustration, consider the utterance: "I'd rather not bring up anything about lunch for now." In existing frameworks, this instruction is ambiguous: it is unclear which memories fall under "lunch," whether the intent is to delete, hide, or demote them, and how long "for now" should last. Different systems therefore behave inconsistently: some temporarily suppress lunch-related items without persistence, others delete them outright, and many simply ignore the command. In contrast, Text2Mem formalizes the request as a `Demote` operation with explicit fields: the `target` is all memories tagged with `lunch`, and the `args` specify a lowered priority rather than deletion. The validator ensures that the action is a demotion, the parser normalizes the scope and priority, and the adapter consistently reduces retrieval weight across backends. This guarantees that everyday expressions map to predictable and safe memory operations (Figure 1).

This paper introduces Text2Mem, a unified memory operation language that addresses the fragmentation of existing systems. Text2Mem provides a compact but expressive operation set aligned with the cognitive stages of encoding, storage, and retrieval. The language eliminates redundancy while elevating advanced controls such as merge, split, promote or demote, lock, and expire to first-class status with precise semantics. Each operation is expressed through a schema-based specification that enforces required fields and invariants, while a type parser produces strongly typed operation objects ready for execution. Together, these features ensure that memory commands are formally defined, automatically validated, and safely instantiated.

Beyond language design, Text2Mem enhances portability, consistency, and research reproducibility. The same typed object can be executed in both a SQL-like reference backend and adapters for real frameworks, ensuring consistent behavior across systems. By separating language understanding from backend execution, Text2Mem provides a stable interface for agents, reduces ambiguity in everyday use, and enables replicable studies of long-horizon memory. In sum, Text2Mem establishes the first standardized pathway from natural language to reliable memory control.

This paper makes the following contributions:

- We propose **Text2Mem**, the first unified memory operation language for LLM-based agents. It defines a compact but expressive set of twelve operations, spanning encoding, storage, and retrieval, with clear semantic boundaries and support for higher-order controls.
- We introduce a **schema-based specification** that formally encodes each operation's required fields, invariants, and constraints, together with a type parser that produces strongly typed operation objects for safe and deterministic execution.
- We demonstrate **portability across execution backends**: the same typed object can be executed in a SQL-like reference backend and mapped to real memory frameworks, ensuring consistent behavior and enabling reproducible studies of long-horizon memory.

2 Background and Motivation

2.1 Agent memory and operation frameworks

Recent work has explored augmenting LLM agents with human-inspired or tool-based memory mechanisms. Early studies draw on theories of human memory, such as hippocampal indexing, to integrate language models with structured stores for more effective consolidation and retrieval [9, 10]. Others make transient context explicit, treating attention caches or hierarchical stores as first-class carriers of working memory, thereby reducing inference costs while maintaining recall [4]. More functionally oriented systems mimic human behaviors such as note-taking or summarization to improve organization and durability [5, 11, 12].

In parallel, tool-based approaches provide explicit interfaces for editing or extending memory. Parameter-level methods expose APIs for inserting, modifying, or deleting knowledge within models [13, 14]. External memory modules mitigate context-window bottlenecks through extract–update workflows or graph-structured representations [8, 15, 16].

Building on these foundations, system-oriented designs aim to treat memory as a first-class operating component. MemGPT proposes modularizing context into dynamic pages [7], A-MEM introduces agentic memory abstractions for LLM agents [17], and MemOS presents a more comprehensive operating system view with richer primitives and scheduling capabilities [6]. These frameworks make memory manipulation more explicit and practically useful, but they largely remain fragmented. They often expose CRUD-style utilities or ad-hoc extensions, and lack a systematic, semantically precise operation language to unify higher-order controls such as prioritization, consolidation, or lifecycle governance.

2.2 Lessons from text-to-SQL

An instructive parallel comes from research on text-to-SQL. This field has faced the same fundamental challenge as memory-centric agents: natural language is underspecified, ambiguous, and highly variable, yet systems must translate it into precise, executable operations.

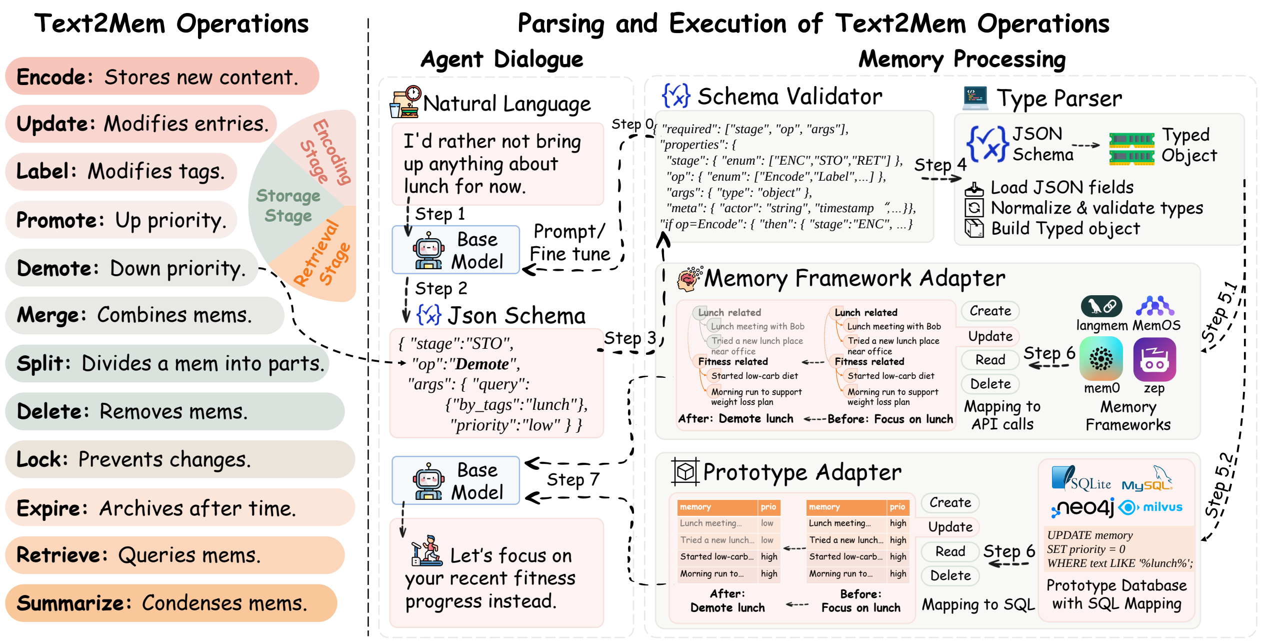
On the modeling side, early systems such as Seq2SQL [18] treated the task as generating SQL queries from utterances. Later work introduced richer architectures, for example RAT-SQL [19] with relation-aware schema encoding, and UniSAr [20] with structure-aware generation that integrates schema and language signals. This evolution marks a shift from surface mappings to models that explicitly encode constraints and structural dependencies.

On the benchmark side, the breakthrough was to introduce a constrained schema layer for standardized evaluation. Spider emphasized cross-domain generalization under a unified schema representation [21], and CoSQL extended this idea to conversational settings with clarification and repair grounded in the same schema

[22]. More recent efforts such as BIRD and LogicCat highlight that larger scale and deeper reasoning only increase the importance of a shared schema backbone [23, 24].

This perspective informs our design. Memory commands today are in the same position SQL queries once were: vague in everyday use, inconsistent across implementations, and lacking a common schema. Text2Mem applies the lessons of text-to-SQL by defining a compact but expressive operation set, enforcing schema-level constraints, and grounding execution in typed objects to make memory control precise and reproducible.

3 Text2Mem



Text2Mem Operations
Encode: Stores new content.
Update: Modifies entries.
Label: Modifies tags.
Promote: Up priority.
Demote: Down priority.
Merge: Combines mems.
Split: Divides a mem into parts.
Delete: Removes mems.
Lock: Prevents changes.
Expire: Archives after time.
Retrieve: Queries mems.
Summarize: Condenses mems.
Encoding Stage
Storage Stage
Retrieval Stage
Parsing and Execution of Text2Mem Operations
Agent Dialogue
Memory Processing
Natural Language
I'd rather not bring up anything about lunch for now.
Step 1
Base Model
Prompt/ Fine tune
Step 2
Json Schema
{ "stage":"STO", "op":"Demote", "args": { "query": {"by_tags":"lunch"}, "priority":"low" } }
Step 0
Schema Validator
"required": ["stage", "op", "args"], "properties": { "stage": { "enum": ["ENC","STO","RET"] }, "op": { "enum": ["Encode","Label",...] }, "args": { "type": "object" }, "meta": { "actor": "string", "timestamp" ...}}, "if op=Encode": { "then": { "stage":"ENC", ... }
Step 4
Type Parser
JSON Schema
Typed Object
Load JSON fields
Normalize & validate types
Build Typed object
Step 3
Memory Framework Adapter
Lunch related
Fitness related
After: Demote lunch
Before: Focus on lunch
Create
Update
Read
Delete
Mapping to API calls
Step 6
langmem
MemOS
mem0
zep
Memory Frameworks
Step 5.1
Step 5.2
Base Model
Step 7
Let's focus on your recent fitness progress instead.
Prototype Adapter
After: Demote lunch
Before: Focus on lunch
Create
Update
Read
Delete
Mapping to SQL
Step 6
SQLite
MySQL
neo4j
milvus
UPDATE memory SET priority = 0 WHERE text LIKE '%lunch%';
Prototype Database with SQL Mapping

Figure 2 Illustration of the Text2Mem execution pathway. Natural language instructions are normalized into memory operation schema instances, which are validated, parsed into typed operation objects, and finally executed through adapters to real memory frameworks or, alternatively, through a SQL-based prototype backend for controlled verification.

To address the limitations of current systems, we introduce **Text2Mem**, a memory operation language that provides a standardized pathway from natural language instructions to reliable execution. As illustrated in Figure 2, Text2Mem follows a three-stage pipeline. First, free-form utterances are mapped into **memory operation schema instances**, expressed in a JSON-based format with explicit fields and constraints. Second, a **validator** checks structural and semantic legality, after which a **parser** converts schema instances into **typed operation objects** that normalize arguments such as time ranges, priorities, and tags. Third, an **adapter** compiles typed objects into concrete actions that can be executed either on a SQL-like prototype database or on the APIs of real memory frameworks. This separation ensures that the same instruction can be translated, verified, and executed consistently across different backends.

The remainder of this section elaborates the core components of Text2Mem: the design of its verb-centered operation set (Section 3.1), the schema specification and typed object model (Section 3.2), and the validator–parser–adapter pipeline that grounds execution across diverse backends (Section 3.3).

3.1 Operation Set Design

A central component of Text2Mem is its verb-centered operation inventory. The design follows three principles. First, the verbs are mutually exclusive: each denotes a unique atomic function without semantic overlap.

Second, the set is complete, covering both basic read–write primitives and advanced controls required in practice. Third, the set is minimal, avoiding redundancy while supporting the full memory lifecycle.

Stage	**Operation**	**Description**	**MemOS**	**mem0**	**Letta**
Encoding	`Encode`	Insert new memory with metadata	✓	✓	✓
Storage	`Update`	Modify existing fields	✓	△	△
	`Label`	Add or edit tags/facets	△	△	△
	`Promote`	Raise priority or set reminders	–	–	–
	`Demote`	Lower priority or archive items	–	–	–
	`Merge`	Combine records with lineage	–	–	–
	`Delete`	Remove via soft/hard deletion	✓	✓	✓
	`Split`	Break composite into units	–	–	–
	`Lock`	Restrict editing rights	–	–	–
	`Expire`	Enforce lifecycle deadlines	–	–	–
Retrieval	`Retrieve`	Run filtered, ranked queries	✓	✓	✓
	`Summarize`	Generate compact summaries	△	△	△

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.

The final inventory, summarized in Table 1, contains one operation for encoding, eight for storage, and two for retrieval. Each operation has a distinct functional scope, ensuring mutual exclusivity while jointly covering the full lifecycle of memory management.

Encoding is represented by a single verb, `Encode`, which introduces new information and subsumes lightweight attention to salient content. Storage is the richest stage, reflecting the diversity of memory management. `Update`, `Label`, and `Delete` provide basic editing. Higher-order verbs extend functionality: `Promote` and `Demote` adjust priority, `Merge` consolidates overlapping entries, `Split` decomposes composite ones, `Lock` freezes sensitive items, and `Expire` enforces lifecycle constraints. Retrieval exposes memory back to the agent. `Retrieve` issues filtered and ranked queries, while `Summarize` condenses content into compact forms for reuse.

Beyond defining the operation set, Table 1 also compares their availability in representative frameworks (MemOS[6], mem0[8], Letta[7]). The contrast reveals a clear pattern: while basic primitives such as `Encode`, `Delete`, and `Retrieve` are universally supported, higher-order controls including `Promote`, `Demote`, `Merge`, `Split`, `Lock`, and `Expire` remain absent. This fragmentation highlights the need for a unified specification like Text2Mem.

3.2 Operation Schema

A central challenge for memory-centric agents is that natural language instructions are underspecified: they omit scope, lifecycle rules, or access constraints that are critical for reliable execution. The operation schema of Text2Mem addresses this gap by providing a formal and executable contract. Each command is instantiated as a JSON object whose explicit fields make implicit assumptions visible and enforceable. In doing so, the schema ensures that memory instructions are not only interpretable but also portable across heterogeneous backends.

Structure and constraints. Every schema instance consists of four global keys: `stage` identifies the cognitive stage (encoding, storage, retrieval); `op` specifies the operation verb; `target` selects the affected memories through IDs, tags, queries, or filters; and `args` provides operation-specific arguments. An optional `meta` field records auxiliary metadata such as actor, language, or timestamp. The schema enforces both structural rules and semantic invariants. Required fields are specified by if–then clauses: for example, `Encode` requires a `payload`, while `Promote` must contain either a `priority`, a `weight_delta`, or a `remind` rule. Cross-field invariants preserve coherence: items marked `locked=true` cannot be hard-deleted, `Expire` must specify a finite horizon via `ttl` or `until`, and `Merge` must preserve lineage rather than overwrite.

Illustrative examples. The expressiveness of the schema is best demonstrated by complex natural language instructions that require decomposition into multiple atomic operations.

- **Conditional prioritization with differentiated permissions.** NL: "Promote all tasks to high priority with weekly reminders, but restrict edits to the owner; subtasks remain append-only for the team."

  ```
  {"stage":"STO","op":"Promote",
   "target":{"by_tags":["task"]},
   "args":{"priority":"high",
           "remind":{"rrule":"FREQ=WEEKLY;BYDAY=MO"}}}

  {"stage":"STO","op":"Update",
   "target":{"by_tags":["task"]},
   "args":{"set":{"write_perm_level":"owner_only"}}}

  {"stage":"STO","op":"Lock",
   "target":{"by_tags":["task","subtask"],"match":"all"},
   "args":{"mode":"append_only",
           "reason":"team allowed to append updates"}}
  ```

- **Cross-modal ingestion with lifecycle demotion.** NL: "Encode this paper PDF as a reference with embeddings, set read access to team and write access to maintainers, and after six months demote it to low-priority archive."

  ```
  {"stage":"ENC","op":"Encode",
   "args":{"payload":{"url":"https://example.org/paper.pdf"},
           "type":"reference","use_embedding":true,
           "tags":["paper","reference"]}}

  {"stage":"STO","op":"Update",
   "target":{"by_tags":["paper","reference"],"match":"all"},
   "args":{"set":{"read_perm_level":"team",
                  "write_perm_level":"maintainer"}}}

  {"stage":"STO","op":"Expire",
   "target":{"by_tags":["paper","reference"],"match":"all"},
   "args":{"ttl":"P6M","on_expire":"demote"}}
  ```

These examples demonstrate how the schema decomposes underspecified utterances into precise atomic actions, while encoding temporal filters, access control, inheritance, and lifecycle policies. By explicitly representing what natural language leaves implicit, the operation schema provides the foundation for deterministic parsing and execution across heterogeneous memory backends.

3.3 Validator–Parser–Adapter Pipeline

Pipeline overview. Once natural language instructions are normalized into schema instances, Text2Mem executes them through a structured pipeline with three components. The validator checks each instance for structural completeness and semantic coherence. The parser transforms validated instances into typed operation objects, normalizing parameters into canonical forms. The adapter then maps typed objects into backend actions, either through the SQL-based prototype or real memory frameworks. All executions return a unified `ExecutionResult` that records status, affected items, and state changes. This design provides three guarantees: malformed instructions are rejected before execution, valid instances are deterministically parsed and executed, and typed objects behave consistently across heterogeneous backends.

Validator. The validator serves as the first safeguard of the pipeline. It checks schema instances against the structural and semantic rules defined in Text2Mem. At the structural level it enforces required fields, allowed values, and type constraints. At the semantic level it inspects invariants across fields, such as prohibiting hard deletion of locked items or requiring finite horizons for expiration. When violations are detected, the validator halts execution and returns a structured error that specifies the failing field and rule. This prevents unsafe or ambiguous commands from entering the system and provides actionable feedback for correction.

Parser. The parser converts validated schema instances into typed operation objects that can be directly stored and manipulated in system memory. Each field is cast into strongly typed variables, ensuring explicit and unambiguous representation. During this process the parser normalizes parameters: absolute and relative time expressions are converted into ISO8601 or RFC5545 forms, priority values and aliases are mapped to a fixed enumeration, and tags are deduplicated. Rules for reminders and updates are expanded into structured objects with clear boundaries. If parsing fails due to incompatible values or unresolved references, execution halts with a parser-level error. The parser thus guarantees that all operations entering execution have consistent types, normalized parameters, and deterministic behavior.

Adapter. The adapter bridges typed operation objects with concrete execution environments. It supports two main execution paths, with optional integration of language model services.

Mapping to memory frameworks. For real-world systems such as MemGPT, mem0, or Letta, the adapter maps each typed object to the corresponding API call. A `Promote` operation with a reminder is translated into a framework-specific priority adjustment and scheduling request, while a `Lock` operation is expressed as a permission update. This translation layer ensures that identical typed objects yield equivalent effects across heterogeneous frameworks.

Mapping to SQL prototype. In the prototype backend, operations are compiled into SQL statements against a lightweight relational schema. For example, `Encode` is compiled into an `INSERT` with text, tags, and embeddings; `Update` becomes an `UPDATE`; `Delete` distinguishes soft and hard deletion via a status flag. Structural operations are also captured: `Split` generates multiple rows with inherited attributes and bidirectional links, while `Merge` consolidates entries under a primary identifier. This SQL pathway provides a transparent and auditable reference implementation for controlled verification and reproducible experiments.

LLM integration. Some operations require language model capabilities beyond symbolic execution. Typical cases include `Encode`, which may depend on embeddings for semantic retrieval, and `Summarize`, which produces compressed representations. In these scenarios the adapter invokes external model services through APIs or local providers such as Ollama. Generated embeddings, summaries, or other derived features are then integrated back into the backend, and results are reported through the unified `ExecutionResult` interface.

Summary. Together, the validator, parser, and adapter form a modular pathway that connects natural language instructions with reliable execution. The validator enforces correctness, the parser guarantees normalized and typed representations, and the adapter ensures consistent action across backends while integrating model-driven capabilities when required. This layered design provides safety, determinism, and portability, establishing a foundation for the system-level implementation described next.

4 Text2Mem Bench (Planned)

Text2Mem Bench is a planned benchmark that evaluates the pathway from natural language instructions to reliable memory control. The goal is to measure both language understanding and execution fidelity under a shared schema and operation set. The present paper focuses on the language and pipeline design. The full benchmark will be released in future work.

Two-layer evaluation. The benchmark separates planning and execution. In the planning layer, the input is natural language and the output is a memory operation schema instance. Metrics include schema validity, slot accuracy, key–value F1, semantic scoring with an adjudicator model, and executability in the prototype backend. In the execution layer, the input is a schema instance and the output is the state change or retrieval result. Checks include state diffs for edits, ranking shifts for promote or demote, trigger correctness for expire, and cross backend consistency between the SQL prototype and real frameworks.

Dataset design. The dataset covers operational, inquiry, conditional, and interactive families of instructions. Each operation type includes direct commands, indirect intents, and multi step compositions. Utterances exhibit stylistic diversity, ellipsis, pronominal reference, and ambiguity. Parameters such as time ranges, tags, priorities, and lifecycles are systematically varied. Negative and adversarial cases include malformed schema and near miss mappings. Construction follows a staged process with seed templates, paraphrasing by models, schema alignment, validation with the Text2Mem schema rules, human sampling, and iterative expansion in multiple languages.

Backends and usage. Two backends are used. The SQL based prototype compiles operations into transparent statements for controlled verification and reproducibility. Real framework adapters map the same typed objects to existing systems to observe practical behavior. The benchmark supports base model testing for schema generation, cross backend comparisons, and automated execution tests with gold schema instances.

Scope and roadmap. The initial release targets a compact dataset that spans all operations with a small number of instances per verb, followed by staged scaling and bilingual support. The benchmark is designed to remain coupled to the Text2Mem schema so that future language updates remain comparable.

5 Conclusion

This paper introduced Text2Mem, a unified memory operation language for agents. The design consists of an operation set aligned with encoding, storage, and retrieval, a schema based specification that enforces fields and invariants, typed operation objects for deterministic parsing, and adapters that execute consistently across a prototype database and real frameworks. The result is a standardized pathway from natural language to reliable memory control with clear guarantees of safety, determinism, and portability. A planned benchmark will evaluate the pathway end to end by separating planning from execution and by comparing behavior across backends. We believe this language layer enables more predictable everyday use and provides a stable basis for future evaluation and system integration.

References

[1] 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, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2025.

[2] 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. A survey on large language model based autonomous agents. Frontiers of Computer Science, 18(6), March 2024.

[3] 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, Yiqiao Jin, Fan Zhang, Xian Wu, Hanqing Zhao, Dacheng Tao, Philip S. Yu, and Ming Zhang. Large language model agent: A survey on methodology, applications and challenges, 2025.

[4] 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. Memory^3: Language modeling with explicit memory. Journal of Machine Learning, 3(3):300–346, 2024.

[5] Jiale Wei, Xiang Ying, Tao Gao, Fangyi Bao, Felix Tao, and Jingbo Shang. Ai-native memory 2.0: Second me, 2025.

[6] 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, Zhen Tao, Junpeng Ren, Huayi Lai, Hao Wu, Bo Tang, Zhenren Wang, Zhaoxin Fan, Ningyu

Zhang, Linfeng Zhang, Junchi Yan, Mingchuan Yang, Tong Xu, Wei Xu, Huajun Chen, Haofeng Wang, Hongkang Yang, Wentao Zhang, Zhi-Qin John Xu, Siheng Chen, and Feiyu Xiong. Memos: A memory os for ai system. arXiv preprint arXiv:2507.03724, 2025.

[7] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2024.

[8] Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory, 2025.

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

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

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

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

[13] Ningyu Zhang, Yunzhi Yao, Bozhong Tian, et al. A comprehensive study of knowledge editing for large language models, 2024.

[14] Ziwen Xu, Shuxun Wang, Kewei Xu, et al. Easyedit2: An easy-to-use steering framework for editing large language models, 2025.

[15] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 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, 2024.

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

[17] Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents, 2025.

[18] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning, 2017.

[19] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 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, July 2020. Association for Computational Linguistics.

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

[21] 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. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii, editors, Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, pages 3911–3921, Brussels, Belgium, October-November 2018. Association for Computational Linguistics.

[22] 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, Vincent Zhang, Caiming Xiong, Richard Socher, Walter Lasecki, and Dragomir Radev. 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, November 2019. Association for Computational Linguistics.

[23] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 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, 2024.

[24] 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. Logiccat: A chain-of-thought text-to-sql benchmark for complex reasoning, 2025.

A Schema Illustration

To complement the formal description in the main text, we also provide a visual excerpt of the Text2Mem schema. Figure 3 shows a representative fragment of the JSON Schema specification that governs the structure, required fields, and invariants for memory operations. This example highlights the case of the `Encode` operation, where the `args.payload` must include either raw text, a URL, or a structured object, and where optional attributes such as tags, facets, and time can be attached. By enforcing these constraints at the schema level, Text2Mem ensures that natural language instructions are normalized into well-formed instances before execution.

```
{
  "title": "Text2Mem IR",
  "type": "object",
  "additionalProperties": false,
  "required": ["stage", "op"],
  "properties": {
    "stage": { "type": "string", "enum": ["ENC", "STO", "RET"] },
    "op": { "$ref": "#/$defs/Op" },
    "target": { "$ref": "#/$defs/TargetSpec" },
    "args": { "type": "object" },
    "meta": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "actor": { "type": "string" },
        "lang": { "type": "string" },
        "trace_id": { "type": "string" },
        "timestamp": { "type": "string", "format": "date-time" },
        "dry_run": { "type": "boolean", "default": false }
      }
    }
  },

  "allOf": [
    {
      "if": { "properties": { "op": { "const": "Encode" } }, "required": ["op"] },
      "then": {
        "properties": {
          "stage": { "const": "ENC" },
          "args": {
            "type": "object",
            "required": ["payload"],
            "additionalProperties": false,
            "properties": {
              "payload": {
                "type": "object",
                "oneOf": [
                  { "required": ["text"] },
                  { "required": ["url"] },
                  { "required": ["structured"] }
                ],
                "properties": {
                  "text": { "type": "string" },
                  "url": { "type": "string", "format": "uri" },
                  "structured": { "type": "object" }
                },
                "additionalProperties": true
              },
              "type": { "$ref": "#/$defs/Type" },
              "tags": { "$ref": "#/$defs/Tags" },
              "facets": { "$ref": "#/$defs/Facets" },
              "time": { "type": "string", "format": "date-time" },
              "subject": { "type": "string" },
              "location": { "type": "string" },
```

•••