OMNE-ENGRAM: A PROGRAMMABLE KNOWLEDGE SUBSTRATE FOR AGENTS WITH GOVERNED SCHEMA EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon agents need knowledge representations that evolve with their domains, tasks and information. Retrieval-augmented generation and memory systems often couple representation and retrieval within application-specific pipelines, leaving each application to coordinate structural revisions, validation and query support. We present OMNE-Engram, a programmable knowledge substrate whose domain schemas and governing contracts (covenants) configure reusable write validation, declaration activation and query admission. We establish its representation capacity through theory and experiments: finite typed relational structures admit lossless encoding under stated assumptions, and reconstructed structures and predefined query results match three heterogeneous sources exactly. Fixed-domain applications approach the strongest baselines across five knowledge-QA and memory benchmarks. Agents driven by three different LLMs further design schemas and covenants and revise them through system checks. Together, these results support OMNE-Engram as a reusable foundation for knowledge applications and governed, agent-driven knowledge construction and evolution.

1 INTRODUCTION

Long-horizon agents carry out tasks through sequences of interdependent decisions and interactions. As execution spans multiple contexts or extends into a continuing stream of tasks, maintaining task state, retaining knowledge and reusing experience become essential (Dong et al., 2026). The evidence these tasks need is often fragmented across independently originated sources, such as chats, profiles, tables and documents, and those sources may conflict (Chai et al., 2026). Retrieval-augmented generation (RAG) retrieves evidence for generation (Lewis et al., 2020), while long-term memory retains information across sessions (Wu et al., 2025).

Different domains and tasks often call for different knowledge representations. Domain analysis connects information organization to specific goals (Hjørland, 2002), and knowledge representation determines which objects and relations are available for reasoning (Davis et al., 1993). Agent applications make such choices: LinearRAG organizes entities, sentences and passages for retrieval (Zhuang et al., 2026); HippoRAG 2 integrates passages into an associative knowledge graph (Jiménez Gutiérrez et al., 2025); EverMemOS consolidates episodic memories into thematic scenes (Hu et al., 2026a). Requirements also change within one domain. Classification research documents revisions that affect the organization and accessibility of existing resources (Tennis, 2012). Representations for agents should therefore fit current tasks and remain revisable.

Existing systems support parts of this maintenance. RAG and memory methods couple construction and retrieval with ongoing content integration, and MemOS manages memory objects with version chains and lifecycle policies (Li et al., 2025b). Programmable frameworks support custom extraction schemas, and versioned databases support schema migration (LlamaIndex, n.d.; TerminusDB, n.d.b). A structural change can affect stored records, applicable constraints and dependent queries. When these dependencies reside in application code, each application must coordinate the change across its write and retrieval paths. The challenge is reusable, system-level support for this coordination.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Agents can also maintain parts of their own knowledge. Voyager generates reusable programs and revises them using execution feedback (Wang et al., 2024); A-MEM constructs linked notes and updates their attributes as information arrives (Xu et al., 2025). To support agents in designing and revising knowledge representations, a system must explicitly describe the knowledge structures, applicable constraints and available query capabilities, and check the changes agents propose. The results provide feedback for agents to revise their proposals or adjust data operations.

We present **OMNE-Engram**, a programmable *knowledge substrate* for agent applications (Figure 1). Each *domain* declares a *schema* defining its types, fields and relations, and a governing contract called a *covenant*. The covenant specifies requirements on the schema and domain operations. Its query facet declares the query capabilities required for particular types and fields. These declarations configure the substrate's storage, validation and query support, allowing applications to choose different knowledge representations while reusing the underlying mechanisms.

External agents can inspect these declarations, construct domain knowledge and revise schemas and covenants, including their query capability declarations, as requirements change. Agents decide what to change; OMNE-Engram checks the changes and manages declaration activation and the preparation of query capabilities. This division of responsibility enables *agent-directed schema evolution* under declared governance requirements.

Contributions. **(1) A programmable knowledge substrate.** We present OMNE-Engram, a system in which applications declare domain-specific knowledge representations, governance requirements and required query capabilities, and the substrate provides the corresponding storage, validation and retrieval support, with hosting resting on a lossless representation guarantee for finite typed relational structures under stated value-domain and resource assumptions (Theorem 1). Agents can design and revise these declarations under governance constraints (Section 3).

(2) Empirical validation of the system's capabilities. Experiments establish structural fidelity on three heterogeneous datasets (Section 4.2), RAG and memory applications that perform close to the strongest baselines on five benchmarks (Section 4.3), and agents' ability to design and revise working domains (Section 4.4).

2 Related Work

RAG and memory systems. LinearRAG uses an entity–sentence–passage graph and two-stage retrieval (Zhuang et al., 2026); HippoRAG 2 combines passage integration with Personalized PageRank for associative retrieval (Jiménez Gutiérrez et al., 2025). EverMemOS groups MemCells into MemScenes using semantic similarity and temporal/profile-conflict conditions, updates profiles and retrieves through scenes (Hu et al., 2026a). A-MEM links notes and selectively updates their contextual descriptions, keywords and tags (Xu et al., 2025). MemOS proposes unified management of plaintext, activation and parameter memories, with access policies and memory-object version histories (Li et al., 2025b). These systems develop knowledge organization and maintenance procedures for answering and memory use. Their updates to graph contents, scenes and memory objects identify what evolves within each application; our comparison also examines revisions to the declarations governing those objects.

Knowledge construction and ontology/schema evolution. LlamaIndex supports schema-constrained extraction and composable graph retrieval (LlamaIndex, n.d.); Graphiti accepts custom entity and edge types and documents additive attribute evolution (Zep, n.d.). Agentic-KGR couples model learning and graph construction with staged schema expansion (Li et al., 2025a). DIAL-KG induces relation and event schemas and assesses candidates before extending the schemas used in subsequent construction (Bao et al., 2026). These approaches address how knowledge representations are configured or generated as information arrives. OMNE-Engram provides persistent declarations and operations through which an external construction policy can install and revise its chosen representation.

Programmable knowledge infrastructure. PG-Schema formalizes property-graph types and constraints (Angles et al., 2023); NoSE selects physical storage schemas according to read–write workloads (Mior et al., 2017). TerminusDB supports custom schemas and versioned storage (Ter-

Table 1: Knowledge-management mechanisms. ✓: supported; △: partially supported; –: not explicitly documented. A/C/I: applications/construction frameworks/infrastructure. Governance columns concern data changes, preservation of domain requirements during schema revision, and rule amendments. Access denotes declaration-driven preparation. *The TerminusDB row includes its companion vector-indexing service VectorLink; its structure checks cover migration consistency.

		Schema	Evolution	Governance checks			Access
	System	Domain	Objects	Data	Structure	Rules	Preparation
A	LinearRAG	–	Graph contents	–	–	–	–
	HippoRAG 2	–	Graph contents	–	–	–	–
	EverMemOS	–	Cells; scenes	△	–	–	–
	MemOS	–	Memory objects	△	–	–	–
C	Graphiti	✓	Graph; attributes	△	–	–	–
	DIAL-KG	△	Graph; schemas	△	△	–	–
I	TerminusDB*	✓	Data; schema; index	✓	△	–	✓
	OMNE-Engram	✓	Data; schema; rules; access	✓	✓	✓	✓

minusDB, n.d.a), with checked schema and data migrations (TerminusDB, n.d.b). Its embedding queries and templates configure indexing inputs; its companion service VectorLink builds commit-specific vector indexes (TerminusDB, 2026; TerminusDB Labs, 2023). These mechanisms provide reusable support for representation, change and access. OMNE-Engram expresses domain types, operational requirements and access needs in schema and covenant declarations, and uses them to configure validation and access preparation. OMNE-Engram's query-admission conditions appear in Appendix B.

Governance of knowledge changes and evolution. MemTxn checks source support and temporal visibility and provides journal-based recovery (Cui et al., 2026); SSGM proposes consistency checks, access-scoped retrieval and log-based reconciliation (Lam et al., 2026). We distinguish three governance questions: whether data changes satisfy current constraints, whether structural revisions preserve declared domain requirements, and whether amendments to those requirements are permitted. TerminusDB supplies consistency and migration checks, and server plugins can add pre-commit checks (TerminusDB, 2026). OMNE-Engram applies schema, predicate and topology checks to supported data mutations, checks schema–covenant conformance before paired activation, and validates covenant amendments on their dedicated path. This connects the governance of knowledge changes with the governance of the rules used to check them. Applications determine the semantic validity of their knowledge.

Table 1 summarizes the compared systems' mechanisms for domain declarations, evolution, governance and access support.

3 System design

OMNE-Engram organizes knowledge into domains, each with a schema, a covenant and a typed knowledge graph (Figure 1). The schema defines types, fields and relations; the covenant specifies requirements on data operations and declaration revisions, together with the required query capabilities. The system uses these declarations to check changes to knowledge and declarations, and maintains the graph and derived access structures, such as indexes, through storage backends. External agents choose how to construct, use and evolve domain knowledge through three interaction points: declaring and revising domains (a), writing and updating knowledge (b), and querying (c).

3.1 Domain declarations and knowledge representation

Knowledge representation involves choices about the entities described and their domain meaning (Davis et al., 1993). We focus on its structural aspect: organizing knowledge through explicit types,

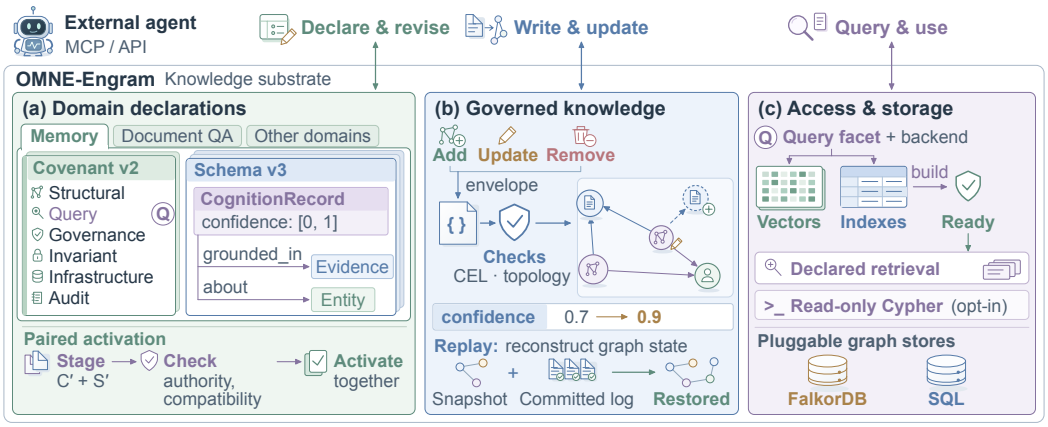

Figure 1: **OMNE-Engram architecture.** (a) A memory domain's covenant (six facets, Section 3.1) and schema excerpt; a candidate pair is checked and activated together (Section 3.2). (b) Checked mutations and replay from a snapshot and the committed log (Section 3.2). (c) Query admission requires the structures prepared from the query facet to be materialized (Section 3.3).

attributes and relations. A domain declares this structure in a schema S and its operating requirements in a covenant C. The schema defines node and edge types, fields, endpoints and structural constraints. A covenant has six facets: structural (types, fields and endpoint types the schema must keep), query (required query capabilities and their target types and fields), governance (write permissions and provenance requirements, and the evolution policy for revisions), invariant (locked fields and values), infrastructure (storage commitments) and audit (revision history). For example, in a project-assignment domain, the schema can represent an assignment as a node linked to a person and a project, and the covenant can require an evidence reference on each write and declare a range-filter capability on assignment dates. Appendix B.2 details the checks and enforcement scope for each facet.

Schema declarations across domains. A fixed meta-schema specifies how any domain declares types, fields, relations and constraints. A document domain can thus organize passages, entities and evidence, while a memory domain organizes events, participants and changing states. All domains change knowledge through typed operations that add or remove nodes and edges or update fields.

Expressiveness for knowledge structures. A finite typed relational structure K over a signature τ has finite sorted carriers, relations with fixed argument sorts and partial attributes (Courcelle & Engelfriet, 2012; Libkin, 2004), whose values lie in an explicit JSON value domain $\mathcal{V}$. We adapt the standard ordered incidence encoding (Courcelle & Engelfriet, 2012, §1.9.1): each object becomes a node, and each relation tuple becomes a separate node whose typed edges identify its argument positions. A three-party assignment therefore keeps its participant roles instead of collapsing into independent pairs. Theorem 1 states that, for every finite signature τ, declarations (S_τ, C_τ) constructed from τ alone and maps Enc_τ and Dec_τ satisfy

$$\mathrm{Enc}_\tau(K) \in \mathcal{L}(S_\tau, C_\tau), \qquad \mathrm{Dec}_\tau(\mathrm{Enc}_\tau(K)) \cong K$$

for every finite τ-structure K, given sufficient identifiers and storage. Here $\mathcal{L}(S, C)$ is the class of finite graphs that logically conform to a schema S compatible with C. Appendix A gives the construction and proof. Thus, domain-specific declarations let OMNE-Engram represent every structure in this class using the same node, edge, payload and cardinality mechanisms.

3.2 Governed Writes and Declaration Revisions

Checks on data operations. Each record that an operation creates or updates is bound to the schema version active when the operation starts, and a write naming a superseded version is rejected. Other records keep their recorded version. Record types, field values and edge endpoints are checked against the bound version, and write permissions and provenance against the covenant active when

the transaction begins. Predicates written in the Common Expression Language (CEL) are evaluated on a record, its incident edges and a neighbourhood of at most two hops, or on the request itself. Topology checks then evaluate declared graph-shape rules, such as degree bounds and acyclicity, on the part of the graph the batch changed, after all its operations are applied and before commit. A rejected write returns *findings*: structured check results that name the rule, its severity, the affected records and a suggested fix. Error-severity findings block the commit (Appendix B.1).

Paired activation. Declarations change through changesets and installs, which submit a candidate schema and covenant together, and through runtime schema operations (Appendix B.2). Every tenant revision must be permitted by the active covenant, never by the candidate, so an agent cannot widen its own authority by proposing a new covenant. The active covenant's evolution policy names who may revise, which parts each may edit and which change classes are allowed; OMNE-Engram computes the classes by comparing the old and candidate declarations. A relaxed requirement also needs a stated rationale and takes effect only in later revisions, and tenants cannot change operator-only controls such as the system lock. A permitted revision must then satisfy $\mathrm{Compatible}(S, C)$: the schema contains every type and field the structural facet requires, with the required field types, and admits every required endpoint type. Once authorization and compatibility checks pass, the candidate schema and covenant become active together in one transaction. If a check fails, the system returns structured findings to guide the agent's revision.

Mutation log and replay. The mutation log records each admitted operation and its execution status, providing a history of knowledge changes. Recovery restores a primary-graph snapshot and reapplies the committed graph operations logged after it in order. Each operation is applied at most once, avoiding duplicate changes during recovery (Appendix B.1).

3.3 Query capability declarations and knowledge access

Access preparation. The covenant's query facet declares the required query capabilities, such as semantic similarity, keyword matching and range filtering, together with the node or edge types and fields they concern. OMNE-Engram matches these requirements against the storage backend's native capabilities. The primary store, which holds the authoritative knowledge, serves some capabilities directly; others need rebuildable derived structures, such as vector collections, keyword indexes or relational projections. Activation triggers their preparation, including a backfill over existing records, and OMNE-Engram records each structure's status.

Query admission. A capability query is admitted only if the active covenant declares that capability for its target and any derived structure it needs is materialized. Materialized means that the backfill over all existing records has completed. Later maintenance is asynchronous, so results from derived structures are marked as candidates to recheck against the primary graph. Undeclared capabilities and unprepared structures return distinct errors, and a failed preparation leaves the declarations active. The query facet declares required capabilities but does not restrict access: graph filters, pattern matches and traversals run on the primary graph without either check.

Read-only Cypher entry. Agents can also query in Cypher, a declarative graph query language, through a separate entry that is disabled by default and enabled by deployment configuration on a supporting backend. The entry accepts only read clauses, runs each query in the backend's read-only mode and returns only whole nodes, edges or paths owned by the caller's tenant. It bypasses query admission and can read every domain in that tenant (Appendix B.1).

3.4 Agent-directed schema evolution

An external agent constructs a domain by declaring its schema and covenant, including the required query capabilities, through the public interfaces. These interfaces also expose a domain's current declarations and available query capabilities. During ingestion and querying, the agent uses incoming information, query results and findings to choose between a data operation and a declaration revision. OMNE-Engram checks both as in Section 3.2 and prepares access as in Section 3.3. Algorithm 2 (Appendix B.2) summarizes how the agent observes domain state, executes operations and revises declarations in response to system feedback.

4 EVALUATION

We ask three questions. (i) Can one set of graph primitives faithfully host structurally different sources, and does OMNE-Engram reject operations that break a hosted domain's declaration? Success means exact agreement of read-back, fixed queries and updates with source references, and rejection of injected faults without rejecting valid operations. (ii) Are applications on fixed OMNE-Engram domains competitive with established RAG and memory systems? Success means scores at or near the strongest baseline under the same answering model. (iii) Can external agents author and revise domains through the governed interfaces, and can a populated domain be revised? Success means that agent-authored domains are activated, populated and queried, with every revision checked by the system, and that each scripted step on a populated domain has its expected outcome.

4.1 EVALUATION SETUP

Benchmarks. Structural hosting uses a Reactome pathway dependency closure (Reactome, n.d.), the complete DrugGeneTargets table of BiomedSQL (Koretsky et al., 2026) and the flight-accident property graph of CypherBench (Feng et al., 2025), with 407 supplementary questions. Applications use 1,000 questions each from HotpotQA, 2WikiMultiHopQA (2Wiki) and MuSiQue (Yang et al., 2018; Ho et al., 2020; Trivedi et al., 2022). Memory applications use 1,382 questions across the ten conversations of LoCoMo-Refined (LoCoMo-R) (mem-eval-suite, 2026; Maharana et al., 2024) and 2,400 questions across five EverMemBench-Dynamic (EMB-D) topics (Hu et al., 2026b).

Evaluation protocol. GPT-5.6-luna and GLM-5.3-Flash answer application questions, and GPT-5.6-luna also answers structural-domain questions; GPT-5.6-sol acts only as a design agent (Section 4.4). Knowledge QA uses exact match (EM) and token-level F1. Memory tasks use accuracy pooled over all questions: Qwen3-14B judges LoCoMo-R, and EMB-D scores multiple-choice answers by its option-letter rule and open-ended answers with Gemini 3 Flash Preview. Scores are percentages, and failed or missing answers count as zero. Answering agents see neither gold answers nor grading feedback. Application scores come from one run without significance testing.

To assess whether applications built on OMNE-Engram are competitive with specialized systems, we select recent methods with leading reported results as strong baselines: LinearRAG (Zhuang et al., 2026) and HippoRAG 2 (Jiménez Gutiérrez et al., 2025) for multi-hop knowledge QA, with Dense RAG as a basic reference; and EverMemOS (Hu et al., 2026a) and MemOS (Li et al., 2025b) for long-term memory. Plain rows answer in one shot, while "+ ReAct" rows (^{R}) and OMNE-Engram use a ReAct-style loop (Yao et al., 2023); on knowledge QA, these agents share one harness and a 24-read ceiling. Every system keeps its construction workflow (Appendices D and E).

4.2 HOSTING AND ENFORCING DECLARED STRUCTURES

Preserving information and operations. For each source, we declare a schema that follows its structure (Table 2, panel A) and map records, attributes and links to nodes, properties and edges by fixed rules. Read-back through the public query interfaces is compared with source identities, types, properties and edge endpoints for every node and edge. Each domain also runs 6 query families with 5 fixed parameter sets each, comparing joins, paths, filters and aggregates with independent source references. All 56 inventory comparisons and 90 queries agree exactly (Table 2, panel B).

On small data copies that retain each full schema, we change a string field, delete and reconstruct an edge, and attempt to add a node with an incompatible field type. Public queries confirm the expected outcome of all 9 checks, including the rejected node's absence.

Enforcement under injected faults. Each domain's declaration gains requirements met by all its source records (Appendix D.1). Each fault breaks one requirement of a valid operation by a minimal edit, such as a mistyped value, a forbidden deletion, a revision dropping a covenant-anchored field, or a query for an undeclared capability. OMNE-Engram rejects 1,800 of 1,800 write faults, 220 of 220 structure-breaking revisions and 70 of 70 query faults (Table 3). Each rejection names the broken rule and edited element or operation kind and leaves the state unchanged. OMNE-Engram rejects none of the 981 valid writes, revisions and queries. Trials of a class apply one rule to dif-

Table 2: Hosting and using three knowledge structures with OMNE-Engram. A: Counts describe the mapped graphs. B: Read-back compares one complete node inventory and one complete edge inventory per relation type; mutation checks use small data copies retaining the full schemas. C: supplementary agent question answering; 13/210/184 questions per column; failures count as zero. Scopes and protocols: Appendix D.

Data source	Reactome	BiomedSQL	CypherBench
A. Source structures and stored graph size			
Source structure	Nested object graph	Relational table	Property graph
# Nodes / # Edges	12,299 / 19,290	43,104 / 110,206	1,683 / 2,212
# Node types / # Edge types	40 / 44	5 / 4	5 / 5
B. Structural and operation checks			
Read-back inventories (matched / total)	45 / 45	5 / 5	6 / 6
Fixed queries (matched / total)	30 / 30	30 / 30	30 / 30
String field update	Pass	Pass	Pass
Edge deletion and reconstruction	Pass	Pass	Pass
Invalid field type rejected	Pass	Pass	Pass
C. Agent answer accuracy (%), GPT-5.6-luna			
SQL/Cypher agent	30.8	50.0	60.3
OMNE-Engram	84.6	59.0	76.1

Table 3: Fault injection on the structural domains, pooled per operation type (classes: Table 7). Cells count operations out of n; parentheses give the lower (faults) or upper (valid operations) bound of the rate's exact 95% interval, in %. Rejected: refused as expected (valid operations: false rejections). Localized: findings name the broken rule and edited element. Unchanged: state intact.

Operations	n	Rejected	Localized	Unchanged
Data writes	1,800	1,800 ($\geq$99.7)	1,800 ($\geq$99.7)	1,800 ($\geq$99.7)
Structure-breaking revisions	220	220 ($\geq$98.3)	220 ($\geq$98.3)	220 ($\geq$98.3)
Queries	70	70 ($\geq$94.8)	70 ($\geq$94.8)	70 ($\geq$94.8)
Valid operations	981	0 ($\leq$0.4)	–	–

ferent records and types, so the intervals describe this run, not independent evidence for other rules. Appendix D.2 reports the latencies of valid and rejected operations.

Supplementary agent QA. As a supplementary check, we let an agent query the three hosted domains through OMNE-Engram and answer questions, testing whether the represented knowledge can support practical tasks. Table 2, panel C, reports answer accuracy, with an agent that directly queries the source databases using read-only SQL or Cypher as a reference.

Answer to (i): yes for hosting, where every check passes. For enforcement, OMNE-Engram rejects every injected fault with localized findings and unchanged state, and none of the 981 valid operations.

4.3 Knowledge QA and memory applications

Manually designed application domains. We build two domains on OMNE-Engram for knowledge QA and long-term memory. The knowledge domain (`engram_knowledge`) organizes corpora into document, passage and claim nodes. Passages link to their parent documents, and extracted claims link to supporting passages, allowing agents to trace retrieved claims to the source text. The memory domain (`engram_cognition`) links cognition records to entities and evidence, and evidence to its sources. Validity intervals and supersession relations represent changes in the recorded knowledge over time. Agents retrieve records through each domain's declared query capabilities and

Table 4: Fixed-domain application quality (%). Bold/underlined: highest/second-highest score per model and metric. ^{R}: + ReAct. Comparison conditions: text and Appendix E.

Knowledge QA (EM / F1)				**Long-term memory**		
Method	HotpotQA	2Wiki	MuSiQue	Method	LoCoMo-R	EMB-D
GPT-5.6-luna				*GPT-5.6-luna*		
Dense RAG	47.30/59.39	27.10/30.34	22.40/28.64	EverMemOS	**78.15**	60.21
LinearRAG	47.50/59.37	43.10/48.87	21.10/26.83	EverMemOS^{R}	<u>76.19</u>	<u>72.71</u>
LinearRAG^{R}	<u>63.90</u>/<u>78.70</u>	<u>71.80</u>/<u>82.13</u>	**52.30/66.10**	MemOS	68.09	59.67
HippoRAG 2	50.60/64.27	36.20/41.10	26.90/34.60	MemOS^{R}	72.14	**74.13**
HippoRAG 2^{R}	63.50/78.01	70.40/80.71	<u>52.10</u>/<u>65.45</u>			
OMNE-Engram	**65.40/78.88**	**74.30/82.85**	50.50/63.95	**OMNE-Engram**	73.52	71.38
GLM-5.3-Flash				*GLM-5.3-Flash*		
Dense RAG	48.20/59.73	33.80/37.03	21.20/27.76	EverMemOS	<u>79.16</u>	53.71
LinearRAG	49.30/60.69	48.90/54.82	20.80/26.39	EverMemOS^{R}	77.93	<u>75.83</u>
LinearRAG^{R}	65.00/79.99	75.90/<u>84.13</u>	<u>55.70</u>/**69.00**	MemOS	69.03	51.42
HippoRAG 2	55.50/68.87	48.00/54.21	29.40/37.78	MemOS^{R}	74.31	75.46
HippoRAG 2^{R}	<u>65.20</u>/**80.29**	**76.60/84.76**	55.00/<u>68.84</u>			
OMNE-Engram	**66.30**/<u>80.23</u>	<u>76.50</u>/83.85	**56.40**/68.23	**OMNE-Engram**	**81.33**	**80.04**

follow graph relations to related content and sources. Each domain reuses one manually designed schema and covenant across its benchmarks, keeping these declarations fixed during evaluation. Figure 3 in Appendix E visualizes the node types, fields and relationships of both application schemas.

Results. On knowledge QA (Table 4), OMNE-Engram scores above both ReAct baselines on HotpotQA and 2Wiki and below them on MuSiQue under GPT-5.6-luna, whereas under GLM-5.3-Flash the three agents score close together and its 2Wiki and MuSiQue F1 are lower. On memory, it scores above every baseline under GLM-5.3-Flash; under GPT-5.6-luna, it trails EverMemOS and EverMemOS^{R} on LoCoMo-R and MemOS^{R} and EverMemOS^{R} on EMB-D. Table 9 in Appendix E reports input tokens summed over the final LLM call of each interaction during construction and answering, to compare the context used in final calls. Under this measure, OMNE-Engram's knowledge-QA answering input is comparable to the ReAct baselines, its construction input is below HippoRAG 2, and its EMB-D answering input is the largest.

Answer to (ii): yes; OMNE-Engram is at or near the strongest baseline in every cell, trailing it by at most 4.63 points (LoCoMo-R under GPT-5.6-luna).

4.4 Agent-driven schema design and governed revision

Revising a populated domain. We test coordinated revisions to fields, covenant requirements and query capabilities on 9 numeric CypherBench fields, one scenario per field (Table 5, Appendix F). Each scenario loads the full graph without its field. A revision then adds the field, a covenant requirement anchoring it and a range-query capability. Explicit writes restore source values, and a final revision drops the still-anchored field. A script, not a design agent, submits these steps. A control query on an existing range path is checked before and after activation and after the writes.

Agent-designed domains and governed revision. From a minimal domain holding only source text, a design agent (GLM-5.3-Flash, GPT-5.6-sol or Claude Opus 5) declares an initial schema S_0 and covenant, including query capabilities, and provides extraction guidance through OMNE-Engram's interfaces. MuSiQue and one LoCoMo-R conversation are each split into two folds; each direction develops on one fold and evaluates on the other. Over three rounds, the agent proposes revisions from development feedback. OMNE-Engram checks each candidate as in Section 3.2 and prepares declared access, and the evaluator keeps a candidate only if its validation correct count increases. Appendix F shows two selected traces of agents acting on a rejection's findings.

Table 5: Revising populated CypherBench domains, steps in order. a/b: a of b over 9 scenarios. First: activation to first admission, median (maximum). Range queries include existing-path checks.

Step	Mechanism checked	Expected outcome	Observed
Adding revision	Paired activation	Accepted	9/9 accepted
Query on the field	Query admission	Refused until materialized	172 refused, 0 answered while not materialized; first 567 (1,937) ms
Existing records	Version binding	Previous version kept	2,830/2,830 kept after activation
Values written back	Explicit migration	Every record written	2,181 records written, 0 rejected
Range queries	Query results	Exact match with source	81/81 exact
Dropping revision	Revision checks	Rejected; state unchanged	9/9 rejected, 9/9 unchanged

Table 6: Agent-designed domains and validation-selected revisions (%). Extraction and answering use GLM-5.3-Flash. Manual domain: the fixed domain of Section 4.3 on the same questions; its LoCoMo-R score is recomputed on conv-26 only, so it differs from Table 4.

Schema author	Version	MuSiQue		LoCoMo-R
		EM	F1	Acc.
Manual domain	Fixed	56.40	68.23	82.61
GLM-5.3-Flash	S_0	47.80	61.52	73.19
	Final selected	49.20	61.57	72.46
GPT-5.6-sol	S_0	48.10	61.61	75.36
	Final selected	48.80	62.14	75.36
Claude Opus 5	S_0	48.40	61.10	73.91
	Final selected	48.80	61.33	76.09

Feasibility and revision outcomes. Across design agents (Table 6), S_0 reaches 61.10–61.61 MuSiQue F1 and 73.19–75.36 LoCoMo-R accuracy. Revision changes MuSiQue F1 by +0.05, +0.53 and +0.23 points for the GLM, GPT and Claude agents, and LoCoMo-R accuracy by -0.72, 0.00 and +2.17 points. The best final configurations remain 6.09 F1 and 6.52 accuracy points below the manual domain.

Answer to (iii): yes, as a feasibility result: every agent's domains are activated, populated and queried, with each revision checked. On populated domains, OMNE-Engram activated 9 of 9 field-adding revisions and rejected 9 of 9 field-dropping ones. It refused 172 queries on the new field as not materialized and answered 0 before materialization. Range-query comparisons, including the existing-path checks, matched the source exactly in 81 of 81 cases.

5 Conclusion

We presented OMNE-Engram, a programmable knowledge substrate for applications and external agents. Schemas define knowledge structures, and covenants declare governance requirements and query capabilities. These declarations configure how the substrate validates graph operations, coordinates revisions and supports queries, allowing applications and agents to define and revise domain-specific knowledge representations.

Our representation result establishes lossless encoding for finite typed relational structures under the stated assumptions. Experiments verify structural fidelity on three heterogeneous sources and show competitive performance for knowledge-QA and memory applications across five benchmarks. Fault-injection tests further demonstrate rejection of operations that violate the tested declaration requirements. External agents also design, populate, query and revise working domains, with system checks providing actionable feedback on proposed changes. OMNE-Engram therefore provides a reusable foundation for domain-specific knowledge applications and agent-driven knowledge evolution.

REPRODUCIBILITY STATEMENT

Appendix A gives the construction, assumptions and decoder. Appendix B gives operational conditions and the interaction algorithm. Section 4 reports the collected results; Appendix F specifies the autonomous design and revision protocol. Appendix D describes the domain mappings and evaluation settings of the structural-hosting experiment, and Appendix E the application comparison. Subsequent comparisons freeze dataset versions, declarations, application policies and budgets before collecting results.

AI USE STATEMENT

Large language models are part of the evaluated systems: they act as domain-design agents, extraction models, answering agents and judges, as specified in Section 4 and Appendices D, E and F. Separately from these experimental roles, we used generative AI tools for implementing methods, including the substrate and the evaluation code; cleaning and reformatting benchmark datasets into domain inputs; formulating mathematical claims and assisting in writing proofs; designing and giving feedback on methodology and experiments; interpreting results; and translation. We have not used generative AI tools for generating synthetic datasets, proposing or refining hypotheses, developing theoretical models or conceptual frameworks, or qualitative or thematic analysis. We also used generative AI tools to write and edit code, create figures, draft and edit sections of the paper, summarize and identify related literature, check and format references, and suggest paper structure. The authors reviewed all AI-assisted work: code was tested, each cited reference was checked against its original publication, proofs were checked step by step, and reported numbers are generated from experiment records rather than typed. The authors take responsibility for the final content of this work.

REFERENCES

Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic, Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic. PG-Schema: Schemas for property graphs. *Proceedings of the ACM on Management of Data*, 1(2):1–25, 2023. doi: 10.1145/3589778.

Weidong Bao, Yilin Wang, Ruyu Gao, Fangling Leng, Yubin Bao, and Ge Yu. DIAL-KG: Schema-free incremental knowledge graph construction via dynamic schema induction and evolution-intent assessment. In Hyungsoo Jung, Tianzheng Wang, Masashi Toyoda, Hyuk-Yoon Kwon, and Jae-woong Lee (eds.), *Database Systems for Advanced Applications (DASFAA 2026)*, volume 16540 of *Lecture Notes in Computer Science*, pp. 357–372. Springer Nature Singapore, 2026. doi: 10.1007/978-981-92-0378-9_22.

Huacan Chai, Yukai Wang, Yingxuan Yang, Dan Peng, Yuanyi Song, Zhihui Fu, Weiwen Liu, Jianghao Lin, Jun Wang, and Weinan Zhang. SMMBench: A benchmark for source-distributed multimodal agent memory, 2026. URL https://arxiv.org/abs/2605.15710.

Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach*, volume 138 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, 2012. ISBN 9780521898331. doi: 10.1017/CBO9780511977619.

Hanshuai Cui, Zhiqing Tang, Zhi Yao, Fanshuai Meng, Qianli Ma, and Weijia Jia. MemTxn: A transaction boundary for source-supported updates and complete-state recovery in agent memory, 2026. URL https://arxiv.org/abs/2607.27834. arXiv:2607.27834v1.

Randall Davis, Howard Shrobe, and Peter Szolovits. What is a knowledge representation? *AI Magazine*, 14(1):17–33, 1993. doi: 10.1609/aimag.v14i1.1029.

Guanting Dong, Xiaoshuai Song, Yuyang Hu, Jiajie Jin, Chenghao Zhang, Yifei Chen, Xiaoxi Li, Huaying Yuan, Xinyu Yang, Tongyu Wen, Jiejun Tan, Hongjin Qian, Shijue Huang, Junting Lu, Zhenyu Li, Wanjun Zhong, Yutao Zhu, Tat-Seng Chua, Zhicheng Dou, and Ji-Rong Wen. Towards long-horizon agents: A survey. *Preprints*, July 2026. doi: 10.20944/preprints202607.1328.v1. URL https://doi.org/10.20944/preprints202607.1328.v1.

Yanlin Feng, Simone Papicchio, and Sajjadur Rahman. CypherBench: Towards precise retrieval over full-scale modern knowledge graphs in the LLM era. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8934–8958, 2025. doi: 10.18653/v1/2025.acl-long.438. URL https://aclanthology.org/2025.acl-long.438/.

Birger Hjørland. Domain analysis in information science: Eleven approaches – traditional as well as innovative. *Journal of Documentation*, 58(4):422–462, 2002. doi: 10.1108/00220410210431136.

Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. Constructing a multi-hop QA dataset for comprehensive evaluation of reasoning steps. In *Proceedings of the 28th International Conference on Computational Linguistics*, pp. 6609–6625, 2020. doi: 10.18653/v1/2020.coling-main.580. URL https://aclanthology.org/2020.coling-main.580/.

Chuanrui Hu, Xingze Gao, Zuyi Zhou, Dannong Xu, Yi Bai, Xintong Li, Hui Zhang, Tong Li, Chong Zhang, Lidong Bing, and Yafeng Deng. EverMemOS: A self-organizing memory operating system for structured long-horizon reasoning. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 45836–45853, San Diego, California, United States, July 2026a. Association for Computational Linguistics. doi: 10.18653/v1/2026.acl-long.2125. URL https://aclanthology.org/2026.acl-long.2125/.

Chuanrui Hu, Tong Li, Xingze Gao, Hongda Chen, Yi Bai, Dannong Xu, Tianwei Lin, Xiaohong Li, Yunyun Han, Jian Pei, and Yafeng Deng. Evaluating long-horizon memory for multi-party collaborative dialogues. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, pp. 9082–9090. ACM, 2026b. doi: 10.1145/3770855.3817589.

Bernal Jiménez Gutiérrez, Yiheng Shu, Weijian Qi, Sizhe Zhou, and Yu Su. From RAG to memory: Non-parametric continual learning for large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 21497–21515. PMLR, 2025. URL https://proceedings.mlr.press/v267/gutierrez25a.html.

Mathew J. Koretsky, Maya Willey, Owen Bianchi, Chelsea X. Alvarado, Tanay Nayak, Nicole Kuznetsov, Sungwon Kim, Mike A. Nalls, Daniel Khashabi, and Faraz Faghri. BiomedSQL: Text-to-SQL for scientific reasoning on biomedical knowledge bases. In *Conference on Language Modeling (COLM 2026)*, 2026. URL https://arxiv.org/abs/2505.20321.

Chingkwun Lam, Jiaxin Li, Lingfei Zhang, and Kuo Zhao. Governing evolving memory in LLM agents: Risks, mechanisms, and the stability and safety governed memory (SSGM) framework, 2026. URL https://arxiv.org/abs/2603.11768. arXiv:2603.11768v2.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, 2020. URL https://arxiv.org/abs/2005.11401.

Jing Li, Zhijie Sun, Zhicheng Zhou, Suming Qiu, Junjie Huang, Haijia Sun, and Linyuan Qiu. Agentic-KGR: Co-evolutionary knowledge graph construction through multi-agent reinforcement learning, 2025a. URL https://arxiv.org/abs/2510.09156. arXiv:2510.09156v1.

Zhiyu Li, Chenyang Xi, Chunyu Li, Ding Chen, Boyu Chen, Shichao Song, Simin Niu, Hanyu Wang, Jiawei Yang, Chen Tang, Qingchen Yu, Jihao Zhao, Yezhaohui Wang, Peng Liu, Zehao Lin, Pengyuan Wang, Jiahao Huo, Tianyi Chen, Kai Chen, Kehang Li, Zhen Tao, Huayi Lai, Hao Wu, Bo Tang, Zhengren Wang, Zhaoxin Fan, Ningyu Zhang, Linfeng Zhang, Junchi Yan, Mingchuan Yang, Tong Xu, Wei Xu, Huajun Chen, Haofen Wang, Hongkang Yang, Wentao Zhang, Zhi-Qin John Xu, Siheng Chen, and Feiyu Xiong. MemOS: A memory OS for AI system, 2025b. URL https://arxiv.org/abs/2507.03724. arXiv:2507.03724v4.

Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, Berlin, Heidelberg, 2004. ISBN 9783540212027. doi: 10.1007/978-3-662-07003-1.

LlamaIndex. Using a Property Graph Index. https://developers.llamaindex.ai/python/framework/module_guides/indexing/lpg_index_guide/, n.d. Official documentation, accessed 11 September 2026.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of LLM agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13851–13870, 2024. doi: 10.18653/v1/2024.acl-long.747. URL https://aclanthology.org/2024.acl-long.747/.

mem-eval-suite. LoCoMo-Refined. Dataset and evaluation software, 2026. URL https://github.com/mem-eval-suite/LoCoMo_refined. Released April 14, 2026; accessed September 15, 2026.

Michael Joseph Mior, Kenneth Salem, Ashraf Aboulnaga, and Rui Liu. NoSE: Schema design for NoSQL applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(10):2275–2289, 2017. doi: 10.1109/TKDE.2017.2722412.

Reactome. Data Model. https://reactome.org/documentation/data-model/, n.d. Official documentation, accessed 11 September 2026.

Joseph T. Tennis. The strange case of eugenics: A subject's ontogeny in a long-lived classification scheme and the question of collocative integrity. *Journal of the American Society for Information Science and Technology*, 63(7):1350–1359, 2012. doi: 10.1002/asi.22686.

TerminusDB. TerminusDB source code, version 12.0.7, 2026. URL https://github.com/terminusdb/terminusdb/tree/57f2093baeafd65e16004e84b7b58e0c5cf72858. Commit 57f2093baeaf; accessed September 14, 2026.

TerminusDB. The Open Source Knowledge Graph Database. https://terminusdb.org/docs/knowledge-graph-database/, n.d.a. Official documentation, accessed 11 September 2026.

TerminusDB. Schema Migration Reference Guide. https://terminusdb.org/docs/schema-migration-reference-guide/, n.d.b. Official documentation, accessed 11 September 2026.

TerminusDB Labs. VectorLink source code, version 0.0.8, 2023. URL https://github.com/terminusdb-labs/vectorlink/tree/832feb011e6f22d50f35fcfd8da94c3510eef8fe. Commit 832feb011e6f; accessed September 14, 2026.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. MuSiQue: Multihop questions via single-hop question composition. *Transactions of the Association for Computational Linguistics*, 10:539–554, 2022. doi: 10.1162/tacl_a_00475. URL https://aclanthology.org/2022.tacl-1.31/.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024. URL https://arxiv.org/abs/2305.16291.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. LongMemEval: Benchmarking chat assistants on long-term interactive memory. In *The Thirteenth International Conference on Learning Representations (ICLR 2025)*, 2025. URL https://arxiv.org/abs/2410.10813.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-Mem: Agentic memory for LLM agents. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 17577–17604. Curran Associates, Inc., 2025. doi: 10.52202/085713-0593.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, 2018. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.03629.

Zep. Custom Entity and Edge Types. https://help.getzep.com/graphiti/core-concepts/custom-entity-and-edge-types, n.d. Official documentation, accessed 11 September 2026.

Luyao Zhuang, Shengyuan Chen, Yilin Xiao, Huachi Zhou, Yujing Zhang, Hao Chen, Qinggang Zhang, and Xiao Huang. LinearRAG: Linear graph retrieval augmented generation on large-scale corpora. In *The Fourteenth International Conference on Learning Representations (ICLR 2026)*, 2026. URL https://openreview.net/forum?id=mCtfkypdm6.

A REPRESENTATION CONSTRUCTION AND SCOPE

This appendix defines the objects in Theorem 1 and proves it. Signatures and structures follow Courcelle & Engelfriet (2012, §1.9.1) and Libkin (2004, §2.1). We adapt the former's positive-arity ordered incidence encoding, adding sort labels, partial attributes, a nullary convention and the value domain below.

Structures and isomorphism. Let $\tau = (\mathcal{A}, \mathcal{R}, \mathcal{F})$ be finite sets of sorts, relation symbols and attribute symbols. Each $R \in \mathcal{R}$ has a profile $(A_{R,1}, \ldots, A_{R,k_R})$ with $k_R \geq 0$, and each $f \in \mathcal{F}$ has a source sort A_f. A finite τ-structure K assigns pairwise disjoint finite carriers A_i^K, relations $R^K \subseteq \prod_{p=1}^{k_R} A_{R,p}^K$ and partial functions $f^K : A_f^K \rightharpoonup \mathcal{V}$. The empty product is $\{()\}$, so a nullary relation is false when it is $\varnothing$ and true when it is $\{()\}$. Structures K and K' are *isomorphic*, written $K \cong K'$, when bijections $h_i : A_i^K \to A_i^{K'}$, one per sort, with union h (well defined because the carriers are disjoint) satisfy two conditions. First, $\bar{a} \in R^K$ exactly when $h(\bar{a}) \in R^{K'}$, for every R and every tuple $\bar{a}$ of matching sorts, with h applied componentwise. Second, for every f and $a \in A_f^K$, $f^K(a)$ is defined exactly when $f^{K'}(h(a))$ is defined, and the two values are then equal.

Value domain. The scalar values are null, false, true, the finite IEEE-754 binary64 numbers and the finite strings of Unicode scalar values. Let $\mathcal{V}_0$ be the least set that contains every scalar value, every finite array of elements of $\mathcal{V}_0$, and every finite object mapping pairwise distinct string keys to elements of $\mathcal{V}_0$. Values are compared structurally: numbers by numeric value, arrays elementwise in order, and objects as sets of key–value pairs. The value domain is $\mathcal{V} = \{v \in \mathcal{V}_0 : |\,\mathrm{json}(v)| \leq \beta\}$, where $\mathrm{json}(v)$ is the canonical UTF-8 JSON encoding of v and β is the platform's fixed byte ceiling for one JSON field value.

Logical graphs and conformance. A *logical graph* G has finite sets of nodes, each with a type and a payload mapping declared field names to values, and of edges, each with a type, a source node and a target node. G contains the records of one domain that are bound to one schema version and have an admitted lifecycle status. Operational metadata, engine-managed payload slots, historical versions and records of other domains are excluded. A *renaming* is an isomorphism of logical graphs: bijections between their nodes and between their edges that preserve types, payloads, sources and targets. For a schema S, *logical conformance* $G \models_\ell S$ is the conjunction of four conditions on the literal stored values. (i) Node typing: every node's type is a node type of S. (ii) Payload typing: every node carries every required field of its type, and every present field holds a value of its declared field type that satisfies the field's declared constraints. Boolean fields take values in $\{\mathrm{false}, \mathrm{true}\}$ and JSON fields in $\mathcal{V}$; the construction uses no other field types. (iii) Endpoint typing: every edge's type is an edge type of S, and its source and target types are admitted endpoints of that edge type. (iv) Declared constraints: every cardinality and topology constraint declared for S, and every declared predicate on records, holds on G. For a covenant C, define

$$\mathcal{L}(S, C) = \{G : G \models_\ell S \ \wedge\ \mathrm{Compatible}(S, C)\},$$

where $\mathrm{Compatible}(S, C)$ is the activation compatibility check of Appendix B.2. It checks S against the structural facet of C only and does not depend on G. Conformance and membership in $\mathcal{L}(S, C)$ are invariant under renaming.

Theorem 1 (Finite structural representation)**.** *For every finite signature τ, one can construct declarations (S_τ, C_τ) from τ alone and maps Enc_τ and Dec_τ such that every finite τ-structure K satisfies*

$$\mathrm{Enc}_\tau(K) \in \mathcal{L}(S_\tau, C_\tau), \qquad \mathrm{Dec}_\tau(\mathrm{Enc}_\tau(K)) \cong K. \tag{1}$$

The construction assumes sufficient identifiers and storage for the finite instance and uses the same node, edge, payload, and cardinality mechanisms for every signature. With $n = \sum_i |A_i^K|$, the graph $\mathrm{Enc}_\tau(K)$ has exactly $n + \sum_R (1 + k_R)|R^K|$ nodes and edges in total, with attribute payloads counted separately.

Proof. Declarations. Choose distinct names for an element type E_i per sort and a tuple type T_R per relation. For each position $p \in \{1, \ldots, k_R\}$, declare a directed edge type $e_{R,p}$ from T_R to $E_{R,p}$, the element type for $A_{R,p}$, and require exactly one outgoing $e_{R,p}$ edge on every T_R node,

with unrestricted incoming cardinality. For each attribute f on A_i, the structured payload of E_i has two required fields with fresh names: a Boolean presence field b_f and a JSON value field v_f. Tuple nodes have empty structured payloads, edges have no required payload, and S_τ declares no other constraint or predicate. Natively these are `NodeType`, `AspectSchema`, `FieldSpec` and `EdgeType` declarations. Each role requirement is a `topology.cardinality.out` constraint with `min=max=1` and `applicable_intents` containing only T_R; its scope includes isolated tuple nodes, so a missing role violates the lower bound. Required deployment metadata and lifecycle defaults are fixed independently of the instance. Let C_τ have an absent structural facet, empty invariant commitments, an empty query facet, and infrastructure, governance and audit facets holding fixed valid metadata. The compatibility check reads an absent structural facet as an empty set of structural requirements, so $\mathrm{Compatible}(S_\tau, C_\tau)$ holds and $\mathcal{L}(S_\tau, C_\tau) = \{G : G \models_\ell S_\tau\}$. Both declarations depend on τ alone.

Encoding. Choose injective identifiers $a \mapsto u_a$ for the carrier elements and fresh, pairwise distinct identifiers $t_{R,\bar{a}}$ for relation tuples and for role edges. Give u_a type E_i for $a \in A_i^K$, with $(b_f, v_f) = (\mathrm{true}, f^K(a))$ when $f^K(a)$ is defined and $(\mathrm{false}, \mathrm{null})$ otherwise. Give $t_{R,\bar{a}}$ type T_R and add $t_{R,\bar{a}} \xrightarrow{e_{R,p}} u_{a_p}$ for every position p. Conditions (i) and (ii) hold because every node has a declared type and all its required fields, each b_f is Boolean, and attribute values and null lie in $\mathcal{V}$. Condition (iii) holds because $a_p \in A_{R,p}^K$ gives u_{a_p} type $E_{R,p}$, and (iv) holds because each tuple node receives exactly one edge per position. Hence $\mathrm{Enc}_\tau(K) \in \mathcal{L}(S_\tau, C_\tau)$. The graph has n element nodes, and each relation contributes $|R^K|$ tuple nodes and $k_R|R^K|$ role edges, which gives the stated count. Presence and value slots add $2\sum_{f\in\mathcal{F}} |A_f^K|$ field occurrences plus the copied values.

Decoding. For $G \in \mathcal{L}(S_\tau, C_\tau)$, take its E_i nodes as the decoded carrier of A_i. Each T_R node t has a unique outgoing $e_{R,p}$ edge per position, whose target $a_p(t)$ has type $E_{R,p}$ by endpoint typing. Define $R^{\mathrm{Dec}_\tau(G)} = \{(a_1(t), \ldots, a_{k_R}(t)) : t \text{ has type } T_R\}$, which for $k_R = 0$ is $\{()\}$ exactly when a T_R node exists. Define f on the element nodes with $b_f = \mathrm{true}$, with value $v_f \in \mathcal{V}$ by payload typing. The decoder is therefore total on $\mathcal{L}(S_\tau, C_\tau)$. Since a renaming preserves node types, role edges and payloads, Dec_τ maps renamed graphs to isomorphic structures. Two choices of identifiers yield encodings related by a renaming, so Equation (1) holds for every choice.

Faithfulness. For an encoded K, $h : a \mapsto u_a$ is a sort-preserving bijection onto the decoded carriers. Each $\bar{a} \in R^K$ has exactly one witness node $t_{R,\bar{a}}$, whose role targets form $h(\bar{a})$, and every T_R node is such a witness. Since h is injective, $\bar{a} \in R^K$ exactly when $h(\bar{a}) \in R^{\mathrm{Dec}_\tau(\mathrm{Enc}_\tau(K))}$, nullary R included, and one role type per position keeps repeated arguments and self-relations intact. Finally, $b_f(u_a) = \mathrm{true}$ exactly when $f^K(a)$ is defined, and then $v_f(u_a) = f^K(a)$, so absence stays distinct from a defined null. Hence h is an isomorphism from K to $\mathrm{Dec}_\tau(\mathrm{Enc}_\tau(K))$. □

Conforming graphs outside the image of the encoding. Whenever τ has a relation or attribute symbol, $\mathcal{L}(S_\tau, C_\tau)$ strictly contains the image of Enc_τ: a conforming G is a renaming of $\mathrm{Enc}_\tau(\mathrm{Dec}_\tau(G))$ exactly when no two T_R nodes of one relation have the same role targets and every v_f whose b_f is false holds null. Otherwise Dec_τ merges duplicate tuple witnesses or ignores values. Set semantics of tuples thus holds for encoded states but is not enforced, because OMNE-Engram's declarations provide no uniqueness constraint over role targets; an application that needs it must avoid duplicate witnesses in its own writes.

Native validation and scope. OMNE-Engram's native checks enforce (i), (iii) and (iv) on the records each batch changes, and they check every b_f as a Boolean without conversion. They restrict v_f only by the byte ceiling β, so they also accept values outside $\mathcal{V}$, such as an integer that binary64 cannot represent. States holding such values pass native validation but lie outside $\mathcal{L}(S_\tau, C_\tau)$, and the theorem makes no claim about them. Its conclusions carry over to stored states given sufficient identifiers and storage in the selected backend and a *value round-trip assumption*, which we make without proof. It states that writing a value of $\mathcal{V}$ to a JSON field, or a Boolean to a Boolean field, and reading it back through the selected wire and storage path returns an equal value. Governance requirements on operations, such as write permissions and provenance, play no role in the theorem; a deployment using another covenant must also meet that covenant's requirements. Authorized operations construct these states under the conditions of Appendix B, writing each tuple node in one batch with its role edges.

B OPERATIONAL CONDITIONS AND AGENT INTERACTION

This appendix specifies the write, query, activation and recovery paths of Section 3. Equations (3)–(6) state necessary conditions that OMNE-Engram enforces on these paths. They specify behaviour rather than proved properties, and lifecycle and backend checks apply in addition. For one domain d, the runtime state at step t is

$$X_t = (G_t, \mathcal{D}_t, \nu_t, P_t), \qquad (S_t, C_t) = \big(\mathcal{D}_t^S[\nu_t^S], \mathcal{D}_t^C[\nu_t^C]\big), \tag{2}$$

where G_t holds nodes, typed edges, payloads and the schema version bound to each record, $\mathcal{D}_t$ stores schema and covenant versions including staged candidates, $\nu_t = (\nu_t^S, \nu_t^C)$ selects the active pair, and P_t gives the status of each derived structure.

B.1 EXECUTING DECLARED WRITES AND QUERIES

Version-bound writes. For a batch $B = (m_1, \ldots, m_k)$, let $\sigma(m_i)$ be the schema version bound to the record that m_i creates or changes. An operation that creates a node or edge, or updates a record, sets $\sigma(m_i) = \nu_t^S$, read when the operation starts. A request naming any other version, such as a superseded one, is rejected, and an unknown or retired version fails as an unknown schema. An update revalidates the whole record against ν_t^S and rebinds it only if it conforms, and activation rebinds no record.

Let $\widehat{C}_B$ be the covenant resolved when the transaction begins. The findings $\mathcal{W}_B(\sigma, \widehat{C}_B)$ of B comprise permission and provenance checks under $\widehat{C}_B$, payload, type and endpoint checks under each $\sigma(m_i)$, and predicate and topology checks. With $\mathrm{Err}(U) = \{v \in U : \mathrm{severity}(v) = \mathrm{error}\}$,

$$\mathrm{CommitWrite}(B) \implies \mathrm{Err}\big(\mathcal{W}_B(\sigma, \widehat{C}_B)\big) = \varnothing. \tag{3}$$

A predicate neighbourhood that exceeds a fixed node ceiling raises an error instead of being truncated. Topology checks cover the changed part of the graph including added nodes, so a node added without a mandatory edge is rejected, and they use the edge-type declarations of the latest registered schema version and every topology rule registered for the schema. A failing checker produces an error finding rather than passing the batch silently. Findings are grouped by rule (Appendix C), and warning and info findings are recorded without blocking. The covenant is not re-resolved at commit, but the transaction aborts if the schema catalog version changes after its first operation.

Declared and available queries. A declared query q requests one capability on a node or edge type under (S, C). OMNE-Engram routes it either to a native index of the primary store, written $p_q = \mathsf{direct}$, or to a derived structure p_q, and

$$\mathrm{Admit}(q) \implies \mathrm{Declared}(q, C) \wedge \big[p_q = \mathsf{direct} \vee P_t(p_q) = \mathsf{materialized}\big], \tag{4}$$

where $\mathrm{Declared}(q, C)$ holds when the query facet of the active covenant declares the capability for the query's target type, and P_t is read at admission. A structure becomes materialized only when the backfill over all existing records of its type completes, and keyword and vector structures also need every current indexing task to finish. A truncated scan or lost rows leave it partial or failed. Readiness is recorded per tenant, schema and structure rather than per schema or covenant version, and it does not imply freshness, because maintenance after later writes runs asynchronously. Results from derived structures therefore carry a watermark and are marked as candidates to recheck against the primary graph, whereas primary-store results are marked authoritative.

Read-only Cypher entry. The entry of Section 3.3 admits one comment-free statement built only from listed read clauses and runs it under a per-query timeout. A scalar, aggregate, property projection or null value fails the whole query, and results above a fixed row ceiling are rejected rather than truncated.

Mutation-log replay. Recovery requires maintenance mode, which suspends ordinary write admission, a snapshot recorded as ready whose stored artefacts pass a check, a single-flight leader lease and a gap-free log after the snapshot's bookmark. Log gaps, undecodable entries and conflicting identifiers stop it with explicit errors. Replay checks only that each entry decodes and names a registered operation kind, and node timestamps, absent from the log, are regenerated. Replay applies committed operations directly to the primary graph without rerunning constraint checks; schema and covenant declarations and derived structures are outside this recovery path.

B.2 DECLARATION REVISION AND AGENT INTERACTION

Covenant facets. Besides the contents given in Section 3.1, the query facet declares access paths, the governance facet holds a system lock and ceremony and versioning policies, and the invariant facet also locks transitions and constraints and lists retired field identifiers. OMNE-Engram enforces the structural facet through $\mathrm{Compatible}$, the query facet through Equation (4), per-operation write permissions and provenance through Equation (3), and locked enumeration values when a field constraint is narrowed. The evolution and ceremony policies and the invariant locks also decide which revisions Equation (5) permits. The versioning policy and infrastructure commitments are recorded and reported but not enforced, although changing either is a revision subject to Equation (5).

Declaration channels. (i) An install previews or installs a complete package of schema, covenant and associated components. Its gates check owner authority, with platform-owned packages requiring an operator, and then Equation (5) against the active pair. Only for a permitted revision do they check $\mathrm{Compatible}(S', C')$ and that every access path's type, participating fields and joined edge types, and every predicate and topology target, resolve in the candidate schema. A compare-and-swap rejects an install if the applied package changed since planning, and a drift check rejects one that would discard schema changes made through another channel since the last install. The gates run again under a row lock inside the activation transaction, which re-reads the active pair. (ii) A changeset, refused for domains founded by an install, stages exactly one candidate schema S' and one candidate covenant C', in either order, and commits them under Equation (6). A rejected commit leaves the changeset open. (iii) Runtime schema operations, available for domains not founded by an install, activate a new schema version at once, retire its predecessor and leave the covenant pointer unchanged. Each is first checked by Equation (5) with $C' = C$. A blocking check rejects any such operation that introduces a violation of $\mathrm{Compatible}$ against the active covenant, and one that narrows a field constraint is also rejected if any stored record of the type violates it, including records bound to older versions. In every channel, a rejected revision returns its findings.

Revision permission. A revision replaces the active pair (S, C) by a candidate pair (S', C'), with $C' = C$ for a runtime schema operation. OMNE-Engram authorizes it with the evolution policy of C, never with that of C', so a candidate cannot grant itself the authority it needs. This policy names the principals $\mathrm{Prin}(C)$ that may revise the domain, the parts $\mathrm{Edit}_C(u)$ of the declarations that each principal u may edit (new types or fields, existing definitions, access paths and each covenant facet), and the allowed and blocked change classes $\mathrm{Allow}(C)$ and $\mathrm{Block}(C)$. An empty $\mathrm{Allow}(C)$ allows every class, and a revision is rejected if an active declaration cannot be read.

OMNE-Engram computes the difference Δ between the two pairs itself, pairing types by identity and fields by stable identifier, and does not rely on a class stated in the proposal. An entry of Δ is additive if it adds a type or field and subtractive if it removes or deprecates one. It is structural if it changes a field's name or constraints or an edge type's endpoints, topology or cardinality, semantic if it changes query capability declarations, and meta if it changes the governance facet, a lock or a structural requirement. An entry can belong to several classes, and an entry of an unknown kind is rejected.

The commitments $\Gamma(C)$ are the structural requirements, locks and retired field identifiers of C, its write permissions and provenance requirements, and its evolution and ceremony policies. C' tightens a commitment when it adds or strengthens it, and relaxes one when it removes or weakens it, for example by dropping a lock, permitting a forbidden operation or waiving an evidence requirement. $\mathrm{Rel}(C, C') \subseteq \Gamma(C)$ collects the relaxed commitments, and $\Delta^\circ \subseteq \Delta$ omits the entries that only tighten a commitment, which therefore need neither a permitted class and part nor a rationale. With ρ the rationale submitted with the revision, a tenant principal u may make the revision only if

$$\begin{aligned}\mathrm{Permit}(u, \rho) \iff\ & u \in \mathrm{Prin}(C) \ \wedge\ \neg\,\mathrm{lock}(C) \ \wedge\ \neg\,\mathrm{reqRev}(C) \ \wedge\ \Pi(C') = \Pi(C) \\ & \wedge\ K \cap \mathrm{Block}(C) = \varnothing \ \wedge\ \big(\mathrm{Allow}(C) = \varnothing \vee K \subseteq \mathrm{Allow}(C)\big) \\ & \wedge\ \mathrm{parts}(\Delta^\circ) \subseteq \mathrm{Edit}_C(u) \ \wedge\ \mathrm{Holds}(C, S, S') \ \wedge\ \big(\mathrm{Paused}(C) \Rightarrow S' \equiv S\big) \\ & \wedge\ \big(\big(\mathrm{Rel}(C, C') \neq \varnothing \vee \mathrm{reqRat}(C)\big) \Rightarrow \rho \neq \bot\big).\end{aligned} \tag{5}$$

Here $K = \mathrm{cls}(\Delta^\circ)$ and $\mathrm{parts}(\Delta^\circ)$ collect the classes and parts of the entries in Δ°. $\mathrm{Holds}(C, S, S')$ requires that S' remove no field that C locks against removal, change no type or default that a lock of

C fixes, admit every locked enumeration value that S admits, keep every locked transition of S and retire no locked constraint. S' must also neither reuse a field identifier allocated in S nor move one to another type, and add no violation of $\mathrm{Compatible}(\cdot, C)$ beyond those of S. $\Pi(C)$ holds the system lock $\mathrm{lock}(C)$, with its reason, author and time, and the agent-managed flag, which are operator-only controls that a tenant cannot change. $\mathrm{Paused}(C)$ holds when C is agent-managed and disallows agent evolution, and $S' \equiv S$ compares schema bodies without version numbers. $\mathrm{reqRat}(C)$ and $\mathrm{reqRev}(C)$ state whether the ceremony policy of C requires a rationale for every revision or requires review. Because OMNE-Engram records no reviews, only an operator can revise a domain whose covenant requires review. When an install founds a domain, no active pair exists, and a tenant's install is permitted unless it sets the system lock.

Because Permit reads the policy, locks and structural requirements of C rather than C', a relaxed commitment still binds S' in the revision that relaxes it and takes effect only later; relaxing a lock or a structural requirement is a meta change that C must allow. The declarer of a covenant thus bounds how far agents may revise the domain, and no candidate covenant can extend that bound. Permission and compatibility are checked separately because $\mathrm{Compatible}(S', C')$ compares only the two candidates: a candidate covenant dropping a field's structural requirement and a candidate schema dropping the field are compatible, yet Equation (5) rejects the pair because the requirement in C still binds S'.

Compatibility. Let the structural facet of C require node types N_C, edge types E_C, anchored fields A_C as triples (y, f, κ) of node type, field and field type, and, for each $e \in E_C$, possibly empty source and target types $\mathrm{src}_C(e)$ and $\mathrm{tgt}_C(e)$. A schema S has node and edge types N_S and E_S, admitted endpoint types $\mathrm{src}_S(e)$ and $\mathrm{tgt}_S(e)$, and payload field types $\kappa_S(y, f)$, undefined when y has no field f. Then

$$\begin{aligned}\mathrm{Compatible}(S, C) \iff\ & N_C \subseteq N_S \ \wedge\ E_C \subseteq E_S \ \wedge\ \forall (y, f, \kappa) \in A_C :\ \kappa_S(y, f) = \kappa \\ & \wedge\ \forall e \in E_C :\ \mathrm{src}_C(e) \subseteq \mathrm{src}_S(e) \ \wedge\ \mathrm{tgt}_C(e) \subseteq \mathrm{tgt}_S(e),\end{aligned}$$

where an undefined $\kappa_S(y, f)$ makes the equality false and an absent structural facet contributes empty requirements. The check depends on the two declarations only, not on graph records. It leaves query-facet references to the install gates, neither compiles nor type-checks predicate expressions, and does not cover invariant locks or write permissions, which Equation (5) covers.

Paired activation. An actor u opens a changeset χ for domain d and stages S' and C'. With $\nu' = (\mathrm{ver}(S'), \mathrm{ver}(C'))$, the commit checks and their effect on the active pointers are

$$\begin{gathered}\mathrm{Commit}(\chi) \Rightarrow \mathrm{Open}(\chi) \wedge \mathrm{Authorized}(u, d) \wedge \mathrm{Permit}(u, \rho) \wedge \mathrm{Compatible}(S', C'), \\ \nu_{\mathrm{after}} = \nu' \text{ if } \chi \text{ commits}, \qquad \nu_{\mathrm{after}} = \nu_{\mathrm{before}} \text{ if } \chi \text{ is rejected.}\end{gathered} \tag{6}$$

$\mathrm{Open}(\chi)$ is checked under a row lock with a compare-and-swap on the changeset status. $\mathrm{Authorized}(u, d)$ is established when χ is opened: a tenant domain accepts its tenant, and a platform domain accepts only an operator. Both pointer updates and the status change share one database transaction. The install channel meets the same condition, with its gates in place of $\mathrm{Open}(\chi)$, and a runtime schema operation meets it with $C' = C$ and without $\mathrm{Open}(\chi)$. $\mathrm{Permit}(u, \rho)$ is the condition of Equation (5) for a tenant principal and holds for an operator.

Activation writes no graph record and, apart from the narrowing check of item (iii), revalidates none. Migration therefore means updating records, which revalidates and rebinds them, or writing new records and, where the covenant permits, removing or superseding old ones. The default permission map forbids node removal, retirement, deprecation, merging, splitting and supersession. A covenant that supplies an explicit map replaces it, and operation kinds it omits are allowed. Access preparation starts after commit, and Equation (4) refuses dependent queries until their structures are materialized.

Activation procedure. Algorithm 1 lists these checks in order for all three channels, and the install channel also evaluates them without locks for a preview. A revision that fails Equation (5) is rejected with the permission findings, and a permitted revision returns the findings of every failed later check together. In-place covenant amendment is rejected, so a covenant changes only through a changeset or an install. Every covenant revision, including an operator's, marks C in its activation transaction

as superseded by C' and stores with C' the rationale and one entry per relaxed commitment, and C' starts a new revision history linked to C by this record. Purging a retired domain deletes its records and declarations, so a domain refounded under its identifier has no predecessor to compare with.

Algorithm 1 Governed declaration activation

Require: Active pair (S, C), empty for a founding install; actor u; candidate (S', C'); rationale ρ

function ACTIVATEPAIR(d, u, S', C', ρ)
  if $\neg\,\mathrm{Authorized}(u, d)$ or the channel is closed to d **then return** a refusal
  Begin a transaction; lock the install receipt of d or the changeset row; re-read (S, C)
  if the state planned against has changed **then** roll back; **return** a conflict
  $F \leftarrow$ CHECKPERMISSION(u, ρ, S, C, S', C') ▷ old policy first
  if $F = \varnothing$ **then** $F \leftarrow$ findings of $\mathrm{Compatible}(S', C')$ and of the channel's other checks
  if F has an error-severity finding **then** roll back; **return** F
  Publish: $\nu \leftarrow (\mathrm{ver}(S'), \mathrm{ver}(C'))$; if $C' \neq C$, record C' superseding C with ρ, $\mathrm{Rel}(C, C')$
  Commit; trigger access preparation for (S', C'); **return** activated
end function
function CHECKPERMISSION(u, ρ, S, C, S', C')
  if u is an operator **then return** $\varnothing$
  if (S, C) is empty **then return** a finding if $\mathrm{lock}(C')$, else $\varnothing$
  if $u \notin \mathrm{Prin}(C)$, $\mathrm{lock}(C)$ or $\mathrm{reqRev}(C)$ **then return** a permission refusal
  $\Delta^\circ \leftarrow$ the entries of Δ that do not only tighten a commitment; $K \leftarrow \mathrm{cls}(\Delta^\circ)$
  $F \leftarrow$ findings of $K \cap \mathrm{Block}(C)$, $K \not\subseteq \mathrm{Allow}(C) \neq \varnothing$ and $\mathrm{parts}(\Delta^\circ) \not\subseteq \mathrm{Edit}_C(u)$
  $F \leftarrow F \cup$ findings of $\Pi(C') \neq \Pi(C)$, of $S' \not\equiv S$ under $\mathrm{Paused}(C)$ and of $\neg\,\mathrm{Holds}(C, S, S')$
  if $\rho = \bot$ and $\big(\mathrm{Rel}(C, C') \neq \varnothing \vee \mathrm{reqRat}(C)\big)$ **then** add a rationale finding to F
  return F
end function

Agent interaction. In Algorithm 2, $\mathrm{obs}(X)$ returns the current declarations, available query capabilities and admitted query results. The external agent's execution environment sets the step limit T and resource allowances $\mathbf{b}$, whose components may specify token, model-call or time limits. Each iteration records usage $\mathbf{c}$ in the same units and deducts it from the remaining allowances. Usage includes observation, action selection and execution, including failed attempts.

Algorithm 2 Agent-directed schema evolution

Require: Domain state X, policy π, permitted inputs, resource allowances $\mathbf{b}$, step limit T

$h \leftarrow \langle\rangle$
for $t = 0, \ldots, T-1$ **do**
  Append permitted inputs and $\mathrm{obs}(X)$ to h; select $a \sim \pi(\cdot \mid h, \mathbf{b})$
  if a finishes the task or does not fit $\mathbf{b}$ **then** charge the usage incurred; **break**
  Execute a: revision (Algorithm 1), write (3), declared query (4) or read-only Cypher query
  Record feedback f and usage $\mathbf{c}$; append (a, f) to h; $\mathbf{b} \leftarrow \mathbf{b} - \mathbf{c}$
end for
return X and the interaction trace

C AN EXAMPLE FINDING

This appendix reproduces an excerpt of the response that rejected one write batch while a LoCoMo-R store was built in the study of Section 4.4. The extraction model had linked a fact to an entity through the `grounded_in` edge type, whose declaration admits only passages as targets. The response carries one finding, grouped under its rule, with the offending edge as a sample subject and a suggested fix. The model's next message reads "That edge to an entity was invalid; resubmitting without it." Table 10 summarizes this case.

In the listing, `...` marks omitted text, and a line longer than 88 characters continues on the next line with two extra spaces of indentation.


```
{
 "blocking_scope": "single_call",
 "context": {
  ...
  "violations": [
   {
    "aggregation_key": "__singleton__:schema.endpoint_intent_not_allowed:534c584e2889",
    "layer": 1,
    ...
    "rule": "schema.endpoint_intent_not_allowed",
    "sample_subjects": [
     {
      "allowed": [["memory", "passage"]],
      "edge_id": "8244b307bbaf79f2.f-pottery-workshop-visit-g3",
      "endpoint": "target",
      "intent": "entity",
      "node_id": "97939c6efec47d80.ent-pottery-class",
      "schema_id": "memory"
     }
    ],
    "severity": "error",
    "suggested_fix": "Use an edge type whose target_intents includes the target node's
      (schema_id, intent) pair.",
    "summary_message": "Edge type 'grounded_in' does not accept target
      (schema_id='memory', intent='entity'); allowed: [('memory', 'passage')].",
    "violation_count": 1
   }
  ]
 },
 "error_type": "EngramValidationViolationError",
 "http_status": 422,
 ...
 "reason_code": "engram.mutation.validation_violation",
 "retry_strategy": "not_retryable",
 "status": "error",
 ...
}
```

D DOMAIN SCHEMAS AND EVALUATION SETTINGS

Three structures on one substrate. Figure 2 shows selected declarations of the three evaluated domains, which preserve source identifiers and provenance with the same node, edge and payload primitives.

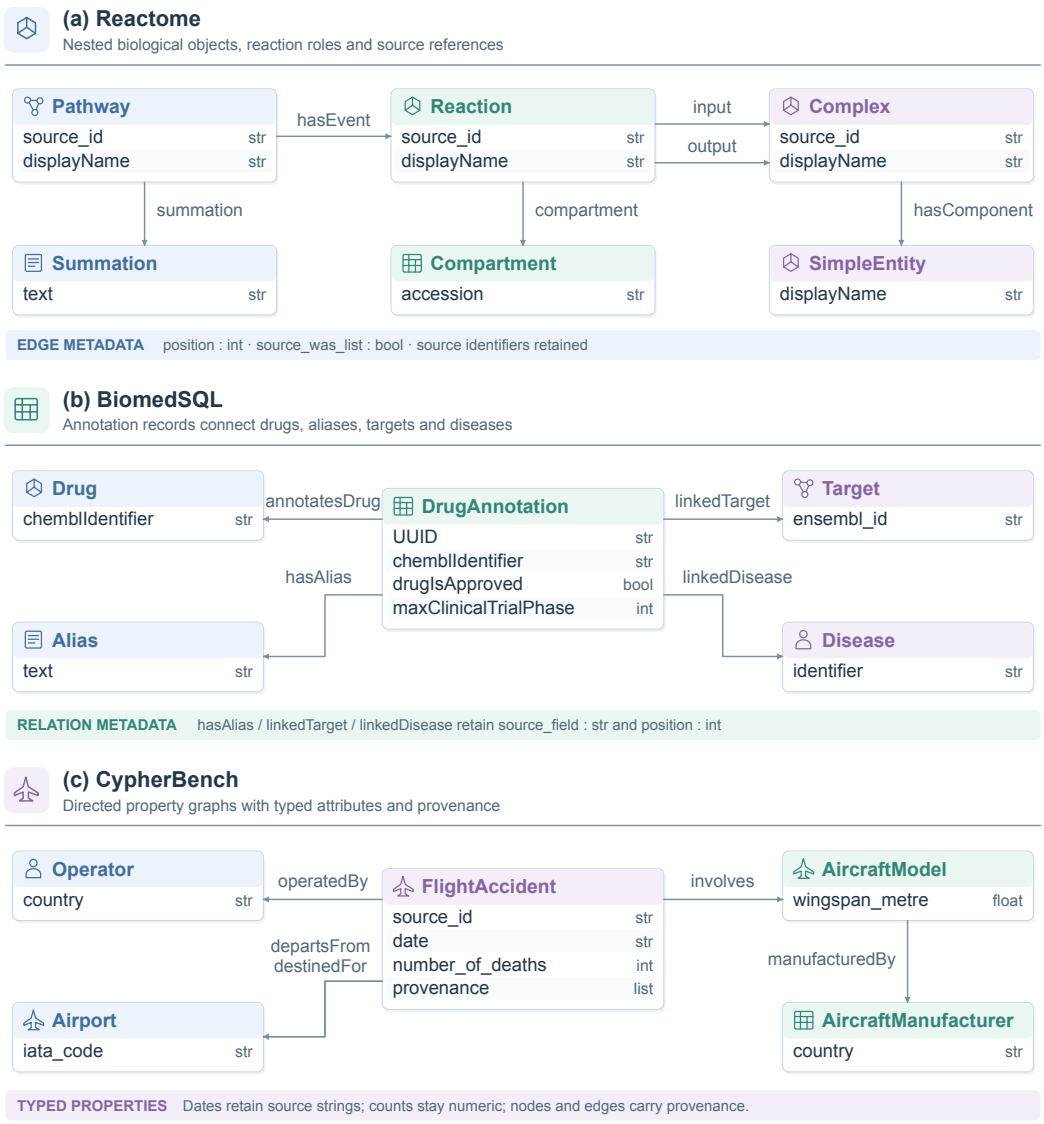

Figure 2: Selected schemas from the three domains hosted by OMNE-Engram. (a) Reactome retains nested objects, reaction roles and occurrence metadata. (b) BiomedSQL preserves each annotation row and connects it to shared entities. (c) CypherBench preserves property-graph types, relationships and provenance. Cards and arrows show selected types, fields and allowed endpoint pairs from the evaluated declarations.

Source mapping and structural checks. Reactome's dependency closure keeps referenced but unexpanded objects as `reference_stub` summaries, and BiomedSQL keeps explicit nulls. The read-back count is the number of inventory groups, one for all nodes and one per edge type, whose identities, types, properties and edge endpoints match the source records. Each domain runs 6 query families with 5 fixed parameter sets per family. Each result is compared with a deterministic reference query over the source data, respecting types, nulls and set, multiset or ordering semantics. The mutation checks change instance data under fixed schemas; the rejected one adds a node with an object value in a declared string field. A separate check hosts the three domains in one deployment and confirms that each retains its inventory.

Agent settings and question selection. Supplementary answering uses GPT-5.6-luna through the OpenAI provider on OpenRouter (snapshot `gpt-5.6-luna-20260709`), with temperature 0, top-$p = 1$, medium reasoning and seed 17. Agents using OMNE-Engram receive the system's built-in help and schema descriptions. The SQL/Cypher agent, with the same model and agent framework, receives the source schema and writes its own read-only SQL (BiomedSQL) or Cypher (Reactome, CypherBench) queries. BiomedSQL and CypherBench answers must agree exactly with typed source results; for Reactome, GPT-5.6-luna judges each answer's fact coverage, accuracy, details and relations against source facts. We retain 184 of 189 CypherBench, 210 of 222 BiomedSQL and 13 of 109 Reactome candidates, excluding those whose source query fails or has an empty reference result, whose query limit leaves ties unresolved, or whose subject lies outside the dependency closure.

D.1 Fault-injection protocol

Declarations and operations. Each domain's evaluated declaration is extended with requirements derived deterministically from the full source data, and an offline check confirms that every source record satisfies them. A field becomes required when every record of its type has a non-null value, and numeric fields are bounded by their observed range. The structural facet requires every node and edge type, anchors every required node field with its type and keeps endpoint types. The governance facet forbids removing, retiring, merging, splitting, deprecating and superseding nodes and requires each write to carry an evidence reference and minimum-length reasoning. The query facet keeps the access paths that need neither an embedding nor a keyword index. With seed 20260925, each domain's base is a connected sample of at most 2,000 nodes, loaded in validated batches. Each valid write submits one held-out record and its incident edges in one batch, citing the source record; held-out records are pairwise non-adjacent, and their edges end in the base.

Each fault applies the minimal edit of its row in Table 7 to one valid operation. Write faults edit one member of a valid write: a bounded value is set just beyond its bound, a moved endpoint lands on a base node, and a provenance fault deletes the intent or only the evidence reference. Some write faults edit the write of a record outside the valid operations; a separate check committed each such write unedited, and all were admitted with their expected effect. A forbidden-operation fault submits, alone, the removal of an existing base node. A structure-breaking revision installs a successor declaration with the covenant unchanged; dropping a required node type also removes or narrows the edge types that reference it. A query fault re-addresses a valid query to an undeclared node type and capability among equality filter, range filter, ordering and aggregation. The 50 undeclared pairs whose capability is keyword search are not run, because keyword and embedding backends lie outside the tested scope. Per class and domain, a seeded draw takes up to 100 faults, or every distinct target when fewer exist, drawing each provenance variant separately. Valid operations are the held-out writes, installs adding one optional field and queries on declared, prepared capabilities. Write faults run before the valid writes.

Outcomes. Expected status, reason code, rule and named elements are fixed per class before the run. A fault is rejected only when the response carries the expected status and reason code; other errors, including server errors, do not count. A rejected fault is localized when its findings or error details identify the expected rule and name the edited element or operation kind. Either the permission or the compatibility check may refuse a structure-breaking revision; either rule counts, and the revision is localized when that check's findings name the dropped or retyped element. A rejected fault is unchanged when database row counts, change-log entries, graph counts and active declaration digests equal their values before submission. A content-level comparison of trials re-

executed on fresh deployments found changes only in a background heartbeat table. A planned trial is excluded before submission if its target state differs from the plan; 0 trials are excluded.

Table 7: Fault classes and valid operations, pooled over domains. d: domains contributing to the row. n: faults or valid operations. Rejected, Localized and Unchanged count operations out of n, defined as in Table 3; parentheses give the lower (faults) or upper (valid operations) bound of the rate's exact Clopper–Pearson 95% interval, in %. For valid operations, Rejected counts false rejections.

Minimal edit	d	n	Rejected	Localized	Unchanged
Data writes					
Value of another JSON type	3	300	300 (≥98.7)	300 (≥98.7)	300 (≥98.7)
Required field deleted	3	300	300 (≥98.7)	300 (≥98.7)	300 (≥98.7)
Numeric value beyond a bound	3	300	300 (≥98.7)	300 (≥98.7)	300 (≥98.7)
Disallowed endpoint type	3	300	300 (≥98.7)	300 (≥98.7)	300 (≥98.7)
Forbidden node removal	3	300	300 (≥98.7)	300 (≥98.7)	300 (≥98.7)
Intent or evidence reference deleted	3	300	300 (≥98.7)	300 (≥98.7)	300 (≥98.7)
Structure-breaking revisions					
Required type dropped, or anchored field dropped or retyped	3	220	220 (≥98.3)	220 (≥98.3)	220 (≥98.3)
Queries					
Undeclared type and capability	3	70	70 (≥94.8)	70 (≥94.8)	70 (≥94.8)
Valid operations					
Held-out record writes	3	900	0 (≤0.5)	–	–
Revisions adding an optional field	3	15	0 (≤21.9)	–	–
Queries on declared capabilities	3	66	0 (≤5.5)	–	–

Base loads and scope. Of the 142 base-load batches, holding 14,025 objects, 0 are rejected. Trials run sequentially from one client. The reported trials do not cover concurrent writers, topology and CEL rules, facets that are recorded but not enforced, revisions through changesets or runtime schema operations, or graph filters, traversals and the read-only Cypher entry, which bypass query admission.

D.2 Latency of governed operations

Unless the caption of Table 8 notes otherwise, each timing is one request's round trip as observed by the client, from sending the request to parsing the response, which covers all server work and database calls but not the stored-state checks before and after it. All timings come from one Linux host, where each domain or scenario ran in its own process with its own PostgreSQL and FalkorDB containers and a server in the client's process, reached over loopback with one request in flight. The 3 domains ran concurrently, the scenarios at most 3 at a time, and the host carried up to 6 such stacks at once because the fault-injection and revision runs overlapped. These are absolute latencies of the governed system, not overhead relative to a configuration without checks, and they indicate magnitude only.

Table 8: Latency of governed operations, in ms, from the fault-injection run and the populated-domain revision run. n: operations timed, as in Table 7 and Table 5. p95: nearest-rank 95th percentile, equal to the maximum when $n < 20$ and omitted (–) for the per-scenario rows. A write returns once committed, before derived structures are updated, and an install returns with the revision already active. The per-mutation row divides each batch's time by its mutations, one for the record's node and one per edge, 2–14 per batch (median 2). Activation to first admission runs from the install response to sending the first admitted range query, polled every 50 ms, so it is an upper bound at that resolution. Migration times the sequential record revisions, one request per record, excluding the preceding field-update attempts; per record divides it by the scenario's records.

Operation	n	Median	p95	Max
Valid operations				
Held-out record write, one batch	900	41	64	456
per mutation	900	18	25	213
Revision adding an optional field, activation included	15	102	636	636
Query on a declared capability	66	21	30	381
Rejected faults				
Data write, fault classes pooled	1,800	15	24	423
Structure-breaking revision	220	32	463	510
Query for an undeclared type and capability	70	11	13	15
Populated-domain revision, per scenario				
Successor install, activation included	9	92	–	128
Activation to first admitted range query	9	567	–	1,937
Migration writes	9	3,920	–	21,800
per record	9	36	–	38

E APPLICATION SCHEMAS, SETTINGS AND TOKEN ACCOUNTING

Application schemas. Figure 3 shows the manually designed schemas used by the two OMNE-Engram applications. The knowledge-QA domain retains documents and passages and links extracted claims to their supporting passages. The memory application uses the cognition domain: cognition records refer to evidence and entities, evidence retains its source, and causal relations have their own typed payloads. The figure shows the three node types of the evaluated knowledge-QA declaration and five memory-related types from cognition. Fields and relationships are selected to make the organization visible.

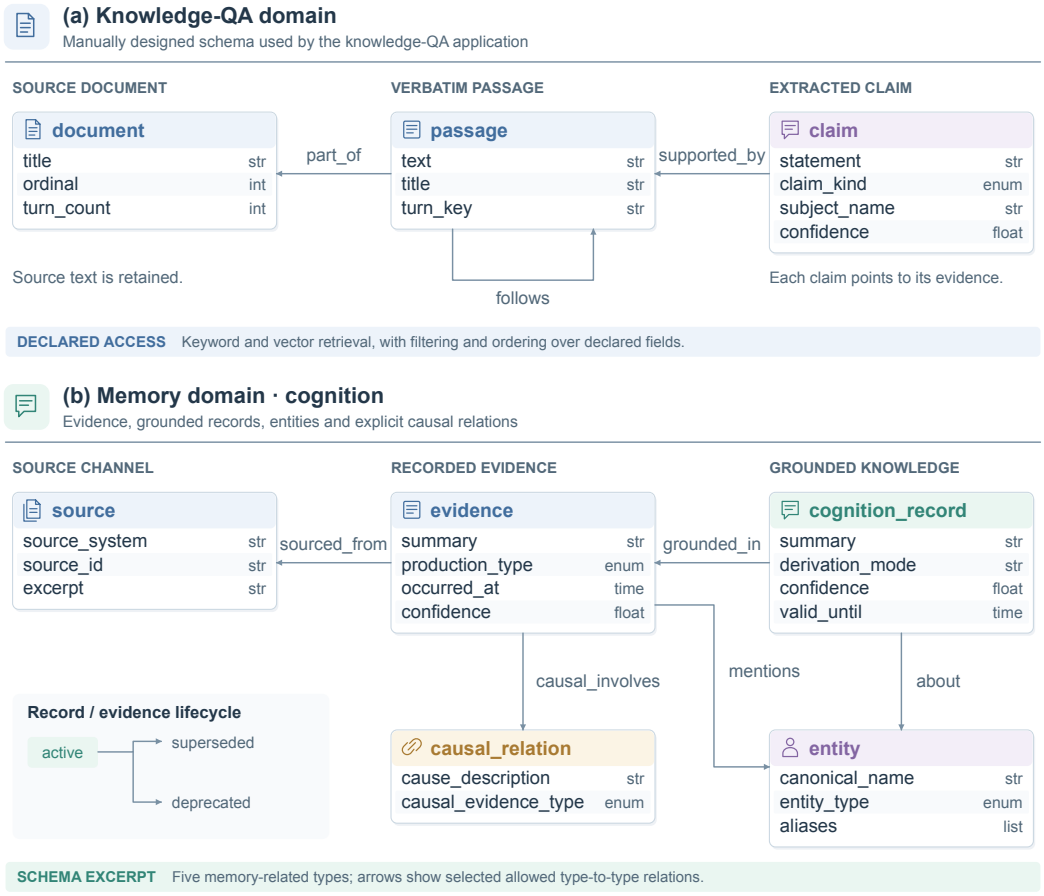

Figure 3: Manually designed application schemas. Cards represent node types and selected fields; arrows name allowed relationships. A loop between ports of one card denotes a relationship between instances of that type. The memory panel shows a selected portion of the cognition domain and the declared lifecycle of records and evidence.

Knowledge-QA settings. The QA corpora contain 9,811, 6,119 and 11,656 passages for HotpotQA, 2WikiMultiHopQA and MuSiQue, respectively. Section 4.1 specifies the question counts, scoring metrics and judge models. QA runs use temperature 0, top-p 1, seed 17, medium reasoning effort and `text-embedding-3-small` as the embedding model. LinearRAG uses `passage_ratio=2` and `top_k_sentence=3`. OMNE-Engram uses the answering model of each run as its extraction model.

Interaction. One-shot QA answers from the top five retrieved passages. Knowledge-QA agents, including OMNE-Engram's, run in Pi, an open-source agent library (`pi-agent-core`) that executes the model's tool calls and returns their results to the model. Baseline agents retrieve up to 20 passages per read from their existing indexes. OMNE-Engram's agent calls the declared operations of its domain with their default result sizes and receives the system's built-in help for these operations and the domain. Memory runs keep each system's recorded prompts, budgets and recovery settings.

Final-call token accounting. Table 9 reports input tokens under one stated measurement choice. For each interaction, we take the recorded input tokens of its last LLM call and sum them over the benchmark: A for answering and B for construction, both in millions. The last call's input contains the prompt, the tool descriptions and all observations accumulated in that interaction, so it measures the context from which the final output is produced. In a fixed extraction pipeline, each standalone LLM call counts as one interaction. Construction totals cover the recorded attempts behind each adopted build. The measure is computed in the same way for all systems and independently of answer correctness.

The measure omits the inputs of earlier calls within a multi-step interaction, all output tokens and all embedding tokens. A last call without recorded usage contributes nothing. Apart from the estimated entries identified in the table caption, each total is therefore a lower bound on the input tokens processed, and none is a billing cost. Dense RAG and LinearRAG use no construction LLM, so their B is zero.

Table 9: Final-call input totals (million tokens): A / B. A: answering; B: construction. ^{R}: + ReAct. EverMemOS/EMB-D B is estimated, excluding profile construction.

Knowledge QA				**Long-term memory**		
Method	HotpotQA	2Wiki	MuSiQue	Method	LoCoMo-R	EMB-D
GPT-5.6-luna				*GPT-5.6-luna*		
Dense RAG	0.77/0	0.87/0	0.84/0	EverMemOS	3.88/14.42	12.98/46.5
LinearRAG	0.76/0	0.77/0	0.77/0	EverMemOS^{R}	10.27/12.57	25.70/46.5
LinearRAG^{R}	8.70/0	10.27/0	13.43/0	MemOS	1.42/2.81	5.46/24.62
HippoRAG 2	0.78/9.35	0.99/5.57	0.87/14.05	MemOS^{R}	5.03/3.17	11.28/24.62
HippoRAG 2^{R}	9.93/9.35	12.34/5.57	14.75/14.05			
OMNE-Engram	9.40/8.47	10.64/5.18	14.84/10.02	**OMNE-Engram**	9.43/5.41	57.33/44.86
GLM-5.3-Flash				*GLM-5.3-Flash*		
Dense RAG	0.78/0	0.89/0	0.85/0	EverMemOS	4.17/18.78	11.88/50.9
LinearRAG	0.78/0	0.79/0	0.78/0	EverMemOS^{R}	6.30/18.10	31.39/50.9
LinearRAG^{R}	8.51/0	9.55/0	16.34/0	MemOS	1.50/2.81	6.07/24.75
HippoRAG 2	0.81/10.60	1.19/6.34	0.90/18.62	MemOS^{R}	3.70/3.21	16.59/24.75
HippoRAG 2^{R}	9.12/10.60	12.38/6.34	15.95/18.62			
OMNE-Engram	8.87/9.15	9.48/5.61	13.99/10.83	**OMNE-Engram**	8.51/6.23	53.44/47.24

F AGENT-DRIVEN SCHEMA DESIGN AND REVISION PROTOCOL

Design and selection. Each design agent starts from the minimal source-record domain, a fixed domain that stores source text as records with keyword and vector search. Using source samples, development questions and OMNE-Engram's interfaces, it declares an initial schema S_0 that adds its own types above these records. It also declares a covenant, retrieval declarations and construction guidance for the extraction model. GLM-5.3-Flash performs extraction and answering throughout. Three revision rounds expose development feedback and permit changes to these declarations and the guidance. OMNE-Engram checks each candidate declaration and each data operation, and the evaluator retains a candidate only when its count of correct validation answers increases. Across the two directions and three design agents, the evaluator retains 8 of 18 revision rounds on MuSiQue and 6 of 18 on LoCoMo-R. These counts describe the evaluator's selection rather than the system's checks; Table 10 shows two selected construction traces in which those checks apply.

Development and evaluation folds. MuSiQue uses 1,000 questions in two folds of 500, each containing 350 feedback and 150 validation questions. LoCoMo-R uses `conv-26`, with two folds of 69, each containing 48 feedback and 21 validation questions. Each direction independently develops on one fold and evaluates on the other, using the metrics in Section 4.1. For feedback questions, the design agent sees predictions, correctness and execution traces. On MuSiQue it also sees aggregate validation scores, and on LoCoMo-R it sees the evaluator's selection decisions. Two held-out MuSiQue questions repeat the text of feedback questions under different identifiers. Removing them leaves the sign of each design agent's F1 difference unchanged.

Endpoint comparison. Table 6 reports standalone frozen runs, in which declarations stay fixed, for S_0 and for the configuration selected after revision. When selection retains S_0, its frozen-run score is reused. A frozen S_0 run and the corresponding revision run start from identical initial declarations, but construction and answering are executed separately in each, so their outputs can differ. The manual domain is the fixed knowledge domain for MuSiQue and the fixed memory domain for LoCoMo-R from Section 4.3. It is scored on the same MuSiQue questions and on the `conv-26` questions of LoCoMo-R, with the same answering model and scoring rules. Its LoCoMo-R accuracy therefore differs from the ten-conversation value in Table 4. Differences between the manual and agent-designed domains concern complete configurations, including construction guidance and retrieval declarations.

Schemas designed and revised by agents. Figure 4 compares two Claude Opus 5 designs at S_0 and their selected revisions. On MuSiQue, the agent introduces `entity` and `claim`; the selected S_3 adds source-title fields and incorporates them into the covenant's query declarations. On LoCoMo-R, it introduces `person`, `episode` and `trait`; the selected S_2 adds `episode.passage_moment`, distinguishing conversation reference time from event time. These cases show the system hosting agent-defined types and subsequent revisions to fields and access declarations. The endpoint scores in Table 6 compare complete construction and retrieval configurations.

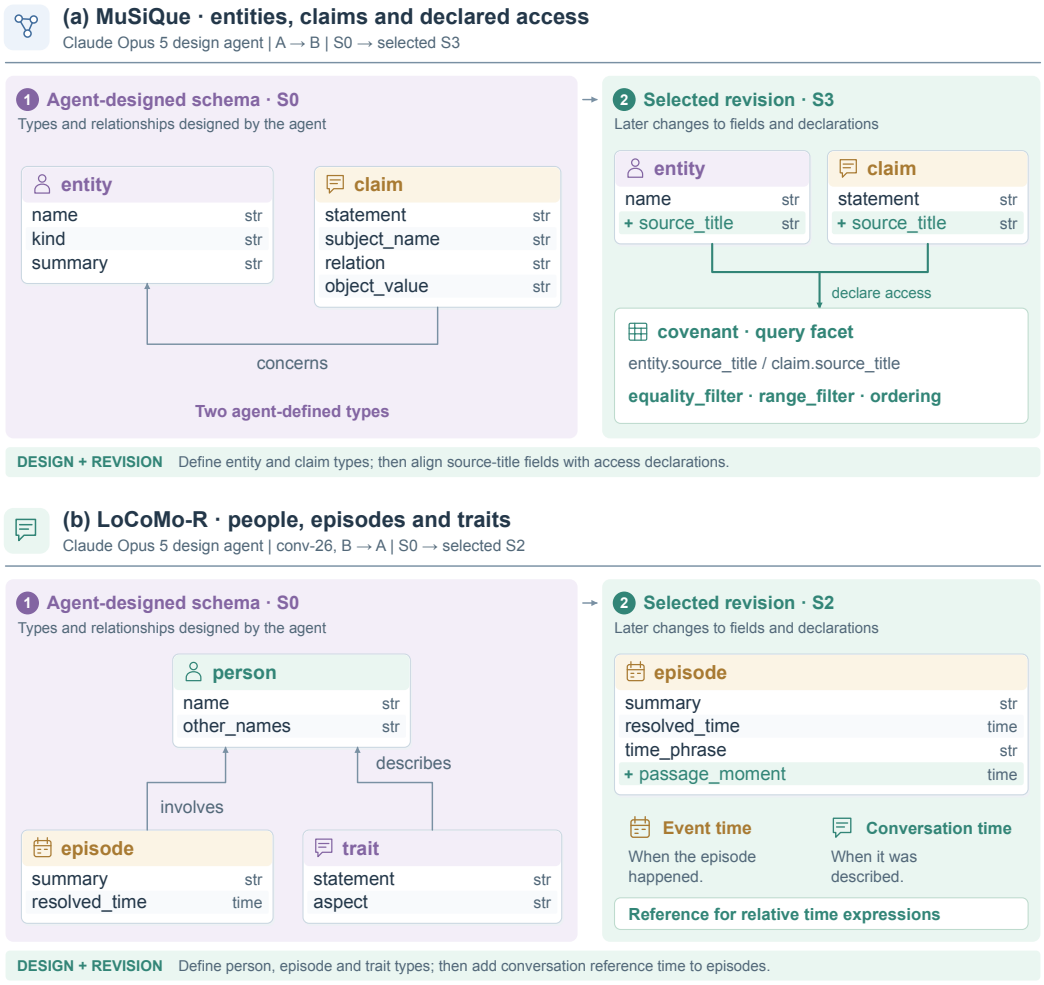

Figure 4: Agent-designed schemas and selected revisions. Both designs use Claude Opus 5: (a) MuSiQue, A→B, selected S_3; (b) LoCoMo-R `conv-26`, B→A, selected S_2. The left panels show agent-defined types in S_0; the right panels show selected later changes. Fields and relations are excerpted; green field rows mark additions after S_0.

Agent operations under governance. Table 10 gives two LoCoMo-R construction traces, selected for illustration rather than as a rate. In the first, the active covenant forbids a node deletion within a batch of writes. In the second, the schema rejects an edge whose relation requires a different target type. In both, the agent drops the operation named in the findings and commits the rest of the batch.

Table 10: Two LoCoMo-R construction traces, selected for illustration.

Agent request	System feedback	Revised request and outcome
Delete an agent-created test node within a batch of writes.	**Covenant permission:** the active covenant's default policy forbids node removal.	Only the deletion is dropped; the remaining writes commit.
Link a fact to an entity through `grounded_in`.	**Schema endpoint rule:** this relation requires a source passage as its target.	The invalid edge is removed, valid passage links are retained, and the corrected batch commits.

Revising populated domains. Each scenario of Table 5 selects a numeric field of one CypherBench node type that holds a value in at least one source record. It runs on a fresh deployment whose configuration enables the relational projections that range queries use. The first declaration extends CypherBench's evaluated declaration with requirements derived from the source data, as for fault injection (Appendix D.1). It removes the selected field from its type's schema, from the covenant's anchored fields and from the type's access paths. The full source graph, without that field's values, is then loaded through the validated batch interface.

The successor declaration restores the field as optional under a new field identifier, anchors it with its source field type and adds a range-filter access path on it. From the install response onward, a script issues a range query on the field between two readiness reads every 50 ms, until the first admission or a 120 s timeout. The table counts a refusal before the first admission when it names the not-materialized reason. It counts an answer as given while not materialized when both readiness reads around the query report the structure as not materialized, so a status change between the two reads is not counted. No loaded record carries the field before migration, so an admitted query before migration returns no record. After the first admission and before migration, the schema version bound to every stored record of the type is read from the primary store.

Migration writes the field's source value into each record that has one, in a request naming the active schema version. The script first submits a field update and, if it is rejected, one record revision that sets the field. The table counts as written each record for which either write committed, and as rejected each record whose explicit migration write was rejected in both forms. Once readiness reports the structure as materialized after these writes, six range queries on the field are compared, as sets of record identifiers, with a reference computed from the source values. Five select values at or above the 10th, 25th, 50th, 75th and 90th source percentiles, and one selects the interquartile range. The six are rerun each second until all match or 30 s pass.

A control query checks an existing range path. It selects values at or above the source median of another numeric field that a range-filter path of the first declaration covers, on the scenario's node type when such a path exists and on another type otherwise. It is compared with its source reference before the successor install, after the first admission and after migration. The table's range queries comprise the six queries and these three checks for every scenario. The dropping revision removes the field from the schema while the successor covenant still anchors it. It counts as rejected when a finding names the type and the field, and the domain state is compared before and after it. Steps after a failed install or a rejected successor revision do not run.