

KAL: Connecting Knowledge Graphs to Geometric Memory Systems

Wing Yan Lau*

Budi Surjanto*

Agus Sudjianto[†]

Abstract

Prior work in this series has developed Geometric Memory Systems (GMS) along three dimensions: a benchmark for measuring structured retrieval [Sudjianto, 2026a], an evaluation framework showing that LLM-as-judge is unreliable on structured tasks [Sudjianto and Lau, 2026b], and a multi-agent architecture in which GMS serves as both persistent memory and governance mechanism [Sudjianto et al., 2026c]. GMS itself supports multiple downstream consumer modes—verification primitive, agent memory, validation harness—but across all of them the knowledge graph that GMS operates over is assumed to exist. This paper addresses the assumption directly: *how should a knowledge graph be designed and exposed so that GMS can consume it cleanly?* We introduce the **Knowledge Adapter Layer (KAL)**—the knowledge plug-in for GMS, and a design specification for the data plane that sits between a knowledge graph (relational store, native graph database, or RDF endpoint) and GMS, regardless of how GMS is used downstream.

KAL has three design components. **The adapter protocol** is a small typed Python interface that any knowledge graph can implement: it groups operations into reads, writes, and similarity search, and pairs them with a capability descriptor that says which subset of the surface a given backend actually supports. The data crossing the protocol boundary—typed nodes, triples, literals, queries, and query results—carries the per-triple metadata GMS consumes (confidence, status, per-engine scores, evaluator identity) as first-class fields rather than opaque payloads. **The federation router** takes multiple adapters and presents them as a single graph to the caller: it fans queries out concurrently, merges results under a backend-independent content-hash dedup rule that handles both node-object and literal-object triples, enforces per-adapter timeouts so one slow source cannot stall the rest, and surfaces partial failures as structured error data so callers can distinguish “adapter returned nothing” from “adapter timed out.” **The schema discipline** is the contract a customer’s knowledge graph must satisfy in order to participate: typed entities with stable identifiers, a controlled predicate vocabulary, edge-attached provenance and verification metadata, and an external-ID registry that lets the same logical entity be addressed through Neo4j node IDs, Postgres UUIDs, or SPARQL URIs without identity collisions across stores. **The protocol is symmetric under a second federation axis:** the same triple can carry verification metadata from multiple GMS instances—a parent agent and its sub-agents, or multiple departmental GMSs within one organisation—and the federation primitives the protocol already exposes serve both axes without surface extension. A reference design grounds the specification: a Postgres adapter against the existing Knowlytix relational schema. The paper closes with **KG autotuning** as a self-improving consumer mode in which GMS’s own geometric signals propose typed mutations to the KG through the same protocol surface—turning KAL-readiness from a one-time onboarding contract into a property the deployment maintains under drift, and letting GMS land as a self-improving knowledge system rather than as a verifier-in-isolation. KAL is released as `knowlytix-kal` under Apache-2.0.

Keywords: knowledge graph adapters, federated retrieval, geometric verification, GMS, LLM hallucination, agentic AI, model risk management

A provisional patent application covering the Knowledge Adapter Layer was filed by KnowlytiX in May 2026. Patent pending.

*Knowlytix.ai

[†]H2O.ai and Center for Trustworthy AI Through Model Risk Management, University of North Carolina at Charlotte

1 Introduction

Three earlier papers in this series develop Geometric Memory Systems (GMS) along complementary axes. **FinStructBench** [Sudjianto, 2026a] provides a measurement instrument for structured retrieval by constructing benchmark instances directly from a document’s knowledge graph. **When the Judge Is Wrong** [Sudjianto and Lau, 2026b] uses that instrument to show that LLM-as-judge accepts 2.9–33.1% of wrong answers on structured tasks. **Policy-Governed Agentic AI** [Sudjianto et al., 2026c] proposes a multi-agent architecture in which GMS serves both as persistent memory and as a governance mechanism: scoring candidate triples for plausibility, filtering through a contradiction gate, and checking path consistency before admitting them to the knowledge graph.

GMS is therefore not a single-purpose engine. The same geometric state can be consumed as a verification primitive against benchmarks, as the persistent memory of a domain agent, or as a validation harness in a model-risk pipeline—the role GMS plays depends on the calling system, not on GMS itself. Across all these consumer modes, however, the *knowledge graph* GMS operates over is assumed to exist—constructed by a benchmark, populated through a Markdown ingestion pipeline, or otherwise pre-supplied. None of the three papers takes a position on how a knowledge graph should be designed, identified, federated, or exposed in order to be a suitable substrate for GMS, regardless of which downstream consumer is doing the calling. The closer GMS gets to production deployment, the more pressing that gap becomes. A bank’s compliance team operates a Neo4j graph of obligations; a research group maintains a Postgres/pgvector index of memos; a clinical workflow queries SPARQL; a fintech runs its primary entity store on a managed cloud relational service (AlloyDB on GCP, Aurora on AWS); an analytics team holds enriched entity data in a lakehouse (Snowflake, Databricks). Each is a real knowledge graph. None is shaped to feed GMS.

This paper presents the **Knowledge Adapter Layer (KAL)**—the data-plane specification that sits between a knowledge graph and GMS. KAL does not introduce new geometric machinery, retrain GMS, or modify any of its operations. It addresses the layer immediately below: *what shape does a knowledge graph need to take, and what contract must it satisfy, for GMS to consume it cleanly—regardless of what the downstream system is doing with the result?*

Scope. This is a design paper. We do not report a new benchmark, a verification accuracy number, or a head-to-head comparison with prior systems. The empirical claims of the prior three papers are not restated. What we offer is the missing specification between those claims and a production deployment, plus a reference adapter design that practitioners can build against directly.

Contributions.

1. **A typed knowledge-adapter protocol.** A single Python Protocol (`KnowledgeAdapter`) exposes a knowledge graph through a focused operation surface—three reads, three writes, and one similarity search—paired with an explicit capability descriptor that declares which subset a given backend supports. The data types that cross the protocol boundary (nodes, triples, typed literals, queries, query results) carry GMS verification metadata as first-class fields rather than opaque payloads.
2. **A federation design with two axes.** A *horizontal* axis unifies many backends into one view (content-hash deduplication that handles both node-object and literal-object triples, per-adapter timeouts, partial failure surfaced as structured error data rather than as silent omission, and documented latency budgets that explicitly account for adapters that fan out an additional LLM call inside the query path). A *vertical* axis unifies many GMS instances over the same view, with one row per (content_hash, verifier) preserving independent

verdicts from a sub-agent hierarchy or a multi-department deployment. Both axes share the same content-hash primitive and the same partial-failure contract; no protocol surface extension is needed.

3. **A schema discipline for GMS-compatible knowledge graphs.** A four-layer specification—entity identity, triple semantics, literal handling, federation identity—with two worked domain examples (finance and clinical) showing that the schema carries across domains.
4. **A reference adapter design.** A Postgres adapter over the platform’s existing relational schema, illustrating a pattern—capabilities declared explicitly, GMS metadata in typed columns rather than opaque blobs, tenancy enforced at the protocol boundary, identity resolved through an external-ID registry—that carries to Neo4j, SPARQL, and other backends as future work.
5. **Autotuning as a self-improving consumer mode.** A feedback loop in which GMS’s own geometric signals propose typed mutations to the KG through the same protocol surface—no protocol extension required—demonstrating that the schema discipline pays for itself twice and that GMS lands as a complete solution rather than as an algorithm-in-isolation.
6. **Open-source release.** `knowlytix-kal` (Apache-2.0): types, protocol, federation router, and the Postgres reference adapter.

The paper proceeds: §2 related work; §3 GMS as consumer of the data plane, including §3.5 on the two federation axes and §3.6 on LLMs across the boundary (as MCP-consumers of KAL, and as builders of KAL through extraction); §4 the protocol surface; §5 federation along the horizontal axis with §5.8 covering the vertical axis; §6 the schema discipline; §7 reference adapter design; §8 KG autotuning as a self-improving consumer mode; §9 limitations; §10 conclusion.

2 Related Work

KAL touches three areas of prior work—RDF/SPARQL federation, LLM retrieval frameworks, and knowledge-graph and provenance standards—without competing with any of them directly. The position KAL occupies is the intersection none of them was designed for: a backend-agnostic protocol whose value envelope is shaped by the needs of a downstream GMS instance.

RDF/SPARQL federation. FedX [Schwarte et al., 2011], ANAPSID [Acosta et al., 2011], SPLENDID [Görlitz and Staab, 2011], and Comunica [Taelman et al., 2018] address federation over SPARQL endpoints with source selection, join-order optimisation, and parallel sub-query execution. Their domain—and their strength—is RDF triple stores. They do not generalise to property graphs or relational stores, and they carry no first-class notion of per-triple verification metadata. KAL’s federation router is simpler than these systems on the query-planning axis (AND-only patterns, no join optimisation) and broader on the backend axis (any store implementing the protocol participates, regardless of its query model). The two are complementary: a SPARQL-side query planner could sit underneath a `SPARQLKnowledgeAdapter` without affecting the rest of the federation.

LLM retrieval frameworks. LangChain retrievers and LlamaIndex provide pluggable abstractions over vector stores, knowledge graphs, and document indices for use inside LLM pipelines. Their primary concern is *retrieval*—surfacing relevant passages or triples for a prompt. The value types they pass typically carry text and an opaque score, not the structured (`confidence`, `verification_status`, `scores`, `verifier`) payload GMS reads and writes. A LangChain retriever could be wrapped as a `KnowledgeAdapter` (read-only, without GMS metadata), and

conversely, a federated KAL view could be exposed as a LangChain retriever for plain-retrieval use cases.

Knowledge graph and provenance standards. W3C RDF [Cyganiak et al., 2014] defines the canonical subject–predicate–object data model and the `xsd:` datatype vocabulary that KAL adopts for typed literals. W3C PROV-O [Lebo et al., 2013] defines the provenance ontology that KAL’s triple-provenance record (source, extracted-at, extractor, source text) aligns with field-for-field. KAL does not require its underlying backends to *be* RDF stores, but it borrows the typed-literal and provenance vocabularies so that RDF-native backends round-trip without translation loss.

3 Background: GMS as a Consumer of the Data Plane

3.1 Geometric Memory Systems

A GMS [Sudjianto et al., 2026c] maintains a geometric state over a typed knowledge graph: entity embeddings, relation transforms registered against a controlled predicate vocabulary, and an exact numerical register with cryptographic integrity for lossless numeric recall. The full algorithmic specification is given in Sudjianto et al. [2026c]; for the purposes of this paper it suffices to note that GMS exposes three operational gates:

- **Plausibility scoring**—given a candidate triple (s, p, o) (subject, predicate, object), GMS returns a score reflecting how well the triple fits the learned geometric structure.
- **Contradiction detection**—given two assertions, GMS returns a contradiction signal that, above a threshold, trips the contradiction gate.
- **Path consistency**—given a path of relations purporting to imply a direct relation, GMS returns a consistency signal.

These three gates have multiple downstream consumers. As a verification primitive they form the `LogicVerifier` used in Sudjianto et al. [2026c], which reaches 100% accuracy on the structured tasks of Sudjianto and Lau [2026b] given a well-formed input graph. As an agent’s persistent memory they back the per-agent domain GMS in the same paper. As a validation harness they underwrite the graph-based retrieval baseline of Sudjianto [2026a]. As a self-improving loop they back the KG autotuning subsystem of §8, which feeds the same geometric signals back into proposals to mutate the KG.

The data-plane requirement is therefore **workload-relative**: KAL-readiness means *fit for the consumer mode you are targeting*, not universally perfect. Different modes invoke different subsets of the gates and tolerate different failure profiles—a verifier flags single-triple inconsistencies in real time; an autotuner aggregates many signals across a batch before proposing a mutation. The four properties of §3.2 are the floor every consumer mode shares; KAL’s purpose is to ensure the input graph satisfies that floor when it arrives at GMS, regardless of which downstream consumer is calling.

3.2 What GMS needs from the data plane

GMS consumes *typed*, *identified*, *GMS-compatible* triples. Four properties of the data plane are load-bearing—and they are properties GMS requires regardless of downstream consumer mode: (1) **typed entities**—GMS’s embedding lookup is undefined on untyped subjects, and the plausibility gate becomes meaningless; (2) **registered predicates**—each predicate in the graph must have a corresponding transform registered on the GMS side, otherwise the path-consistency gate is silently undefined; (3) **stable identity**—GMS’s runtime integer index fragments under ID churn, making outputs order-dependent across calls; (4) **first-class GMS**

metadata—GMS reads and writes per-triple `confidence`, `verification_status`, `verifier`, and per-engine `scores`, and cannot persist its judgements when metadata is stripped at the protocol boundary.

The schema discipline of §6 enforces these properties and the protocol of §4 transports them. This is why KAL is not interchangeable with a generic retrieval-framework abstraction: a generic retriever optimises for relevance and treats per-result metadata as opaque, while KAL’s contract is that the metadata is structured, typed, and consumed by GMS.

3.3 Separation of concerns: GMS vs. KAL

GMS and KAL are two layers with a clean dividing line. GMS owns the geometric algorithm and its internal state—the learned embeddings, the registered relation transforms, the numerical register, and the three operational gates of §3.1. KAL owns the layer immediately below: the protocol that exposes a knowledge graph to GMS, the federation router that combines multiple graphs into one logical view, and the schema discipline that any participating graph must satisfy. The two layers are replaceable independently. A different GMS-class consumer could read the same KAL data plane, and a different data plane could feed the same GMS, provided the four properties of §3.2 hold at the contract boundary.

3.4 Prerequisite: triple-shaped input

This paper assumes the customer holds knowledge in triple-shaped form (s, p, o) , however that shape is realised in their backing store—a Neo4j property graph, a Postgres relational schema (including managed cloud variants such as AlloyDB on GCP and Aurora on AWS), an RDF endpoint, an analytics warehouse or lakehouse (Snowflake, Databricks), or any other store whose native model an adapter can map onto the `KALTriple` envelope. Document-to-triple extraction is an upstream concern, deferred to a separate paper in this series currently in preparation; the chain is only as strong as its weakest link, and KAL is silent on the link above it. KAL’s contract begins where typed triples already exist: the §6 schema discipline shapes them, the §4 protocol exposes them, and the §5 federation router combines them across stores. One narrow exception (§6.6 on bag-of-strings stores) lets a non-triple store participate as a `search_similar_nodes` provider only, with the consumer type-validating results before passing them to GMS—a forward-looking property of the protocol surface, not a demonstrated path in v1. Sources that are not yet triple-shaped at all are not turned away—§7.3 catalogues the three adapter patterns (schema projection, query-time synthesis, pre-extraction upstream) by which an adapter brings such a source into the protocol envelope.

3.5 Two federation axes

The §3.3 separation-of-concerns argument leaves room for *multiple* GMS-class consumers to share one data plane; §3.1 left it implicit. We make it explicit here, because production deployments—particularly in regulated enterprises—rarely have only one. A bank running GMS-enabled validation harnesses across Credit Risk, Market Risk, and Compliance ends up with three GMS instances and a partially shared knowledge graph; a multi-agent system in the style of [Sudjianto et al. \[2026c\]](#) ends up with a parent agent’s GMS coordinating with several sub-agent GMSs over overlapping triple sets. Both deployments are real, and both are the same problem: more than one GMS instance needs to coexist on top of the same data plane without colliding.

The protocol surface is symmetric under a second federation axis. The horizontal axis (the original §5) is identified by `KALTriple.adapter`—the field that names *which backend* a triple came from when many backends federate into one logical view. The vertical axis is identified by `VerificationMetadata.verifier`—the field that names *which GMS instance* authored a verdict when many GMSs federate over the same triple set. The two fields are dual: one indexes the data plane, the other indexes the verdict plane, and the content-hash dedup primitive of §4.3 keeps them coordinated. The federation router (§5) handles both axes with the same machinery;

§5.8 spells out the verdict-plane mechanics.

Two worked deployments make the axes concrete. The **sub-agent hierarchy** is vertical-only: a parent agent’s GMS and several sub-agent GMSs share one logical KG view; each sub-agent writes verdicts under its own `verifier` identity ("`gms:agent-sub-finance`", "`gms:agent-sub-clinical`", ...); federation returns one row per (`content_hash`, `verifier`) pair, so the parent sees all sub-agent verdicts side-by-side and applies its own coordination policy. The **multi-department bank deployment** is both-axes: Credit Risk, Market Risk, and Compliance each own a private KG *and* run their own GMS, but all three also draw on a shared customer-master KG; the horizontal axis unifies private-plus-shared backends per department, while the vertical axis surfaces all three departments’ verdicts on shared-KG triples so the bank-wide validation harness can compose them. Figure 1 sketches the topology.

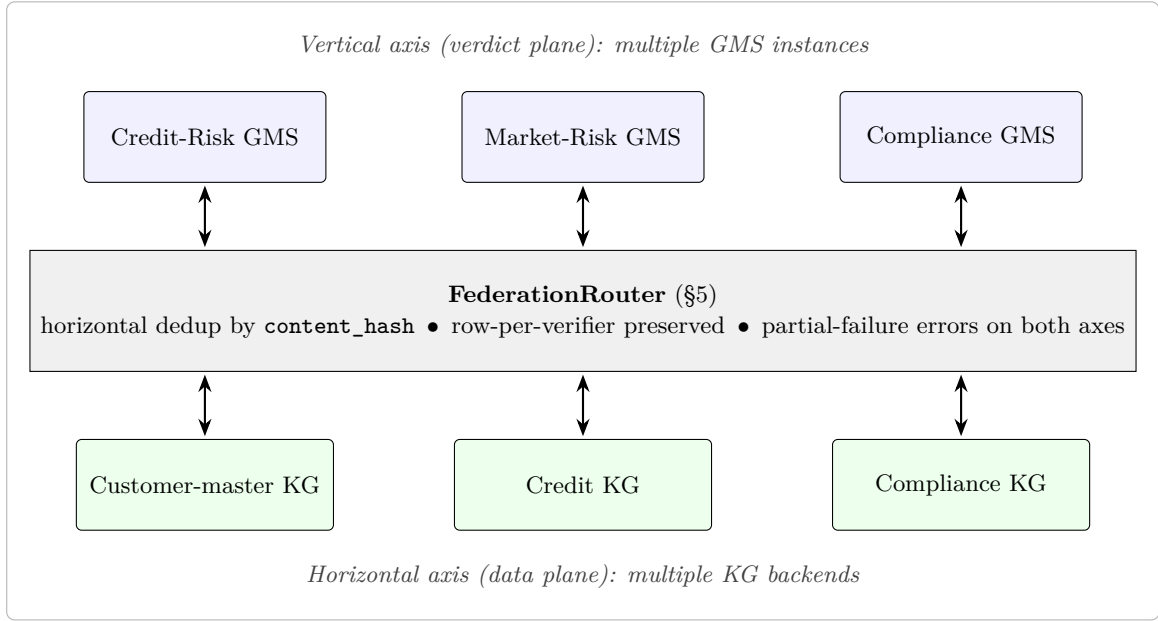


Figure 1: Two federation axes. Arrows are bidirectional throughout: queries and writes flow down (GMS → router → KG); triples and verdicts flow up (KG → router → GMS). The horizontal axis unifies many backends into one view via content-hash deduplication (the §5 federation router; Figure 2 shows its single-axis runtime topology in detail). The vertical axis preserves one row per (`content_hash`, `verifier`) so independent GMS verdicts coexist on the same logical triple. Both axes share the §5 partial-failure contract and the §4.3 typed value envelope.

The architectural payoff is that **the same protocol primitives serve both axes without extension**. `KALTriple` already carries the two identity fields; `KALQueryResult.errors` already classifies partial failure; `ConflictStrategy` already names a default (`ALL`) that is the safe choice when verifiers are not directly comparable (§5 discusses why). What §5 develops for the horizontal axis carries to the vertical axis with no new types, no new methods, and no breaking changes—the spec already supports this topology; this paper makes it explicit.

3.6 LLMs across the boundary

The protocol surface is GMS-shaped but not GMS-exclusive—two LLM relationships sit either side of the same boundary, and KAL is silent on the LLM internals on either side.

LLMs as direct consumers of KAL. A `KnowledgeAdapter` is straightforwardly exposable as a tool over the Model Context Protocol (MCP) [Anthropic, 2024]: `query_triples`, `get_node`, `query_adjacent_triples`, and `search_similar_nodes` become typed tool calls that an LLM agent invokes inline during inference, with the federated KAL view standing in for what would otherwise be ad-hoc retrieval. The LLM then consumes the data plane in the §3.1 sense,

substituting paraphrastic generalisation for the geometric gates GMS would apply; the four §3.2 properties matter for the same reason—untyped, unstable, or metadata-stripped triples ground an LLM no better than they ground GMS. The two consumer modes compose naturally: an LLM agent calls a KAL-backed MCP server for retrieval, then routes structured candidates through GMS for verification before responding. §5 federation operates identically over MCP-side and GMS-side calls. One MCP-specific discipline applies: `tenant_id` is bound by the MCP host from the session context, *not* taken from the LLM-controllable tool-call arguments, so a model that emits a tool call naming another tenant’s identifier cannot cross the tenant boundary (§4.3 expands the threat model).

LLMs as builders of KAL. In the other direction, populating a GMS-compatible KG from source documents is itself an LLM-heavy task. The Knowlytix platform leverages two open-source extractors (`kg_gen` [kg-gen Project, 2025] and `cognee` [Cognee Project, 2024]) that drive LLMs through structured-output prompts to emit typed triples; the resulting triples pass through the §6 discipline at ingestion time and reach GMS by way of any conformant adapter. The full design—extractor architecture, schema enforcement, evaluation against the §6 properties—is the subject of a separate paper in this series currently in preparation. For KAL the relevant invariant is narrow: regardless of which LLM-driven extractor produced a triple, it crosses the §4 protocol boundary in the same typed envelope, and the schema discipline is enforced at the same layer. §7.3 catalogues the three places extraction can sit relative to the adapter boundary (pre-extracted upstream, query-time inside an adapter, or schema-projected from a structured source).

Neither direction extends `KnowledgeAdapter`: the MCP-consumer path adds tools to an inference loop; the extraction path adds populated triples to a store. The protocol surface stays stable in both cases, and the same horizontal-axis federation (§5) and vertical-axis verifier identity (§5.8) machinery applies when an LLM-as-consumer deployment sits alongside, or replaces, a GMS-as-consumer one.

4 The KAL Protocol

The protocol exposes a knowledge graph through a single Python `Protocol` (`KnowledgeAdapter`) plus a small value envelope. Two constraints govern every choice: the adapter implementer should not need to know what GMS is, and the GMS-side caller should not need to know which backend served a triple.

4.1 Capability declaration

Adapters vary widely—a Postgres adapter supports reads, writes, and vector search; a public SPARQL endpoint may be read-only; a third-party-tool-wrapped retrieval source may serve only similarity search. A single frozen `AdapterCapabilities` record handles the variance:

```
@dataclass(frozen=True)
class AdapterCapabilities:
    can_read: bool = True
    can_write: bool = False
    can_delete: bool = False
    supports_vector_search: bool = False
    supports_text_search: bool = False
    supports_provenance: bool = False
    supports_verification_metadata: bool = False
    supports_atomic_writes: bool = False
    max_batch_size: int = 1000
```

Two properties of this descriptor matter. First, every instance is **immutable** (the `frozen=True` flag on the Python dataclass): once an adapter declares its capabilities at construction time they cannot be mutated, so the federation router is free to read, cache, and compare them across concurrent calls without taking a lock. Second, the defaults are **pessimistic**: every flag except `can_read` defaults to `False`, so an adapter must explicitly opt in to writes, deletes, vector search, text search, provenance, verification metadata, and atomic writes. The failure mode for an adapter that forgets to declare a capability is therefore “silently unavailable”—the router skips it for that operation—rather than “invoked and broken at runtime.”

4.2 The protocol surface

```
@runtime_checkable
class KnowledgeAdapter(Protocol):
    @property
    def name(self) -> str: ...
    @property
    def capabilities(self) -> AdapterCapabilities: ...

    # Read
    async def query_triples(self, query: KALQuery,
                           tenant_id: str | None = None) ->
        KALQueryResult: ...
    async def get_node(self, node_id: str,
                      tenant_id: str | None = None) -> KALNode | None
        : ...
    async def query_adjacent_triples(self, node_id: str,
                                     predicate: str | None = None,
                                     direction: str = "outgoing",
                                     tenant_id: str | None = None,
                                     limit: int = 100) -> list[
        KALTriple]: ...

    # Write (guarded by capabilities)
    async def insert_triples(self, triples: list[KALTriple],
                           tenant_id: str | None = None) -> int: ...
    async def update_verification(self, triple_id: str,
                                 verification: VerificationMetadata,
                                 tenant_id: str | None = None) ->
        bool: ...
    async def delete_triples(self, triple_ids: list[str],
                           tenant_id: str | None = None) -> int: ...

    # Search (guarded by capabilities)
    async def search_similar_nodes(self, query_vector: list[float],
                                  tenant_id: str | None = None,
                                  limit: int = 10,
                                  ) -> list[tuple[KALNode, float]]:
        ...
```

The verb convention is load-bearing: `get_*` returns a single item or `None` (the only “not found” channel); `query_*` returns a collection by pattern; `search_*` returns a ranked list under similarity; `insert_*` / `update_*` / `delete_*` are CRUD with integer counts so callers can tell partial from full success. The protocol is mypy-clean with no `Any` types in its declared surface.

The `tenant_id` parameter. Every operation on the protocol takes an optional `tenant_id`. This is the multi-tenant isolation key: in a SaaS deployment serving multiple customers from one platform, the caller passes the requesting customer’s tenant identifier, and adapters that store tenant-partitioned data scope every read and write to rows belonging to that tenant. Adapters whose underlying backend has no notion of tenancy (a single shared SPARQL endpoint, for example) may ignore the parameter; adapters whose backend *does* (the Postgres reference adapter) raise a `TenantRequiredError` when called with `None`. Keeping the parameter on every method—even where some backends will ignore it—means a single tenant filter is impossible to forget at any of the adapter boundaries the data crosses.

When KAL is exposed through an LLM-facing surface (the §3.6 MCP-consumer path), one further discipline is load-bearing: **`tenant_id` is bound by the calling host from out-of-band session context—not surfaced as an LLM-controllable tool-call argument.** An MCP server fronting `KnowledgeAdapter` resolves the tenant from the authenticated session and

injects it before dispatch; the tool schema exposed to the model omits the field. The threat this addresses is concrete: a prompt-injected agent that could specify `tenant_id` in its own tool-call arguments would read across the federation by construction, defeating every per-adapter check downstream. Treating tenant context as session-derived rather than argument-derived is what makes the per-method parameter safe in the presence of adversarial prompts.

4.3 The value envelope: typed boundary data

By *value envelope* we mean the set of Pydantic data types that cross the protocol boundary—what an adapter takes as input and returns as output. The envelope, not the operation signatures alone, is what makes GMS metadata first-class: every triple carries provenance and verification as nested typed records rather than as opaque dictionary payloads. The core types are sketched below; remaining helper types are described in prose afterwards.

```
class KALLiteral(BaseModel):
    value: str # lexical form, e.g.
        "1998" or "394300000000"
    datatype: str | None = None # XSD-style URI, e.g. "
        xsd:integer", "xsd:decimal"
    language: str | None = None # BCP 47 tag for
        language-tagged strings, e.g. "en"

class KALNode(BaseModel):
    id: str # backend-native handle:
        UUID, integer, URI, etc.
    name: str
    node_types: list[str] = [] # one or more controlled
        -vocabulary type tags
    normalized_name: str | None = None # dedup key (see Sec.
        6.1)
    properties: dict[str, object] = {} # adapter-specific
        metadata
    adapter: str | None = None # stamped by the
        federation router

    @property
    def primary_type(self) -> str | None:
        return self.node_types[0] if self.node_types else None

class TripleProvenance(BaseModel):
    source: str | None = None # document URL, agent
        name, etc.
    extracted_at: datetime | None = None
    extractor: str | None = None # e.g. "kg_gen", "cognee"
        ""
    source_text: str | None = None # original text span the
        triple was derived from

class VerificationMetadata(BaseModel):
    confidence: float | None = None # in [0.0, 1.0]
    verification_status: str | None = None # "verified" | "
        unverified" | "disputed"
    verified_at: datetime | None = None
    verifier: str | None = None # agent/system that
        produced the verdict
    scores: dict[str, float] | None = None # per-engine sub-scores
        (one entry per engine)
```

The composite type that ties them together:

```
class KALTriple(BaseModel):
    id: str | None = None          # adapter-local
        identifier
    subject: KALNode
    predicate: str                 # relation name from
        controlled vocabulary
    object: KALNode | None = None  # set when the object is
        a node
    object_literal: KALLiteral | None = None # set when the object is
        a typed value
    provenance: TripleProvenance | None = None
    verification: VerificationMetadata | None = None
    properties: dict[str, object] = {} # edge-level metadata (
        qualifiers, weights)
    adapter: str | None = None      # successful origin
        after federation
```

Several points are worth drawing out from this surface.

Node identity is a string. `KALNode.id` is `str` to accommodate Postgres UUIDs, Neo4j integer IDs, SPARQL URIs, and any backend-native handle. Non-uniqueness across adapters is handled separately, through normalised-name dedup (`content_hash`, below) and the external-ID registry (§6.4).

Object is exactly one of node or literal. Separate optional fields (`object`, `object_literal`) with a Pydantic model-validator that fails construction if both, or neither, is set. RDF flattens these into a single slot; KAL keeps them apart so node-vs-literal handling is a type-level distinction.

Provenance and verification are nested typed records, not free-form metadata. Provenance aligns field-for-field with W3C PROV-O for RDF round-trip; verification carries what GMS writes back through `update_verification` (confidence, status, per-engine sub-scores, verifier identity). Both records survive across adapter boundaries in federated results, so a triple judged by GMS retains its judgement on re-query.

Verifier identity is a stable string. Each `VerificationMetadata` carries exactly one verifier in the `verifier` field, a deployment-stable identifier (`"gms:credit-risk-prod"`, etc.) so verdicts from the same GMS instance group reliably across queries. Under the vertical axis of §3.5, federation produces one `KALTriple` row per (`content_hash`, `verifier`) pair—no list-valued metadata, no breaking change. A typed `VerifierIdentity` record is reserved for v2 (§9).

Queries are AND-only triple patterns plus optional modifiers. A `KALQuery` carries a list of `KALTriplePatterns` (subject/predicate/object each optionally wildcard) plus `text_search`, `vector`, `limit`, `offset`, `min_confidence`, `include_unverified`, and `timeout_ms`. Patterns combine under AND only—OR, NOT, and SPARQL-style variable binding are out of scope for v1. GMS handles multi-hop reasoning geometrically through its own relation transforms rather than query-side joins: KAL retrieves; GMS verifies. Literal-object triples are reachable by constraining subject and predicate; filtering literals by datatype is reserved for v2.

Query results carry partial failure inline. `KALQueryResult` returns triples, a `query_time_ms` wall-clock, and an `errors` list of `AdapterError` records—the only signal the caller has that the result is degraded. `AdapterError` is a data model (not an exception), carrying the adapter name, an `error_type` from the closed set `"timeout"/"auth"/"connection"/"capability"/"circuit_open"/"internal"`, and a message. Single-adapter calls raise from the `KALError` hierarchy directly; the `errors` list is populated only by the federation router (§5).

content_hash: the federation dedup primitive. Every `KALTriple` exposes a derived `content_hash` tuple federation uses to recognise that two triples returned by different adapters are the same logical assertion: $(s_{\text{norm}}, p, \text{"node"}, o_{\text{norm}})$ for node-object triples, $(s_{\text{norm}}, p, \text{"lit"}, v, d, \ell)$ for literal-object triples. Normalised names fall back to `name`; missing `datatype` or `language` are coerced to `"` (not `None`) for stable comparison across Python versions and serialisation paths. The `"node" / "lit"` discriminator is load-bearing—without it, a node named `"$394.3B"` and a literal value `"$394.3B"` would hash identically and federation would silently collapse them.

4.4 What the protocol does not include

Three deliberate omissions. **No transaction surface.** `KAL` exposes no `begin/commit/rollback` operations. **Transactional semantics are the responsibility of the underlying datasource, not of KAL**—each adapter satisfies the contract by guaranteeing whatever atomicity its backing store provides per call. `supports_atomic_writes` advertises the only atomicity guarantee `KAL` itself promises: that a single `insert_triples` batch either persists fully or rolls back fully, within one adapter. Cross-call and cross-adapter transactionality does not generalise across heterogeneous backends and is delegated to the customer’s application layer. **No schema-introspection method**—adapters declare *capabilities*, not the entity types or predicates they hold; schema discovery is delegated to out-of-band tooling. **No event stream**—the protocol is request/response only; change feeds and subscriptions are reserved for v2.

5 Federation Design

A customer with one knowledge graph implements an adapter and calls it. A customer with several—the realistic case—needs the union of those graphs to look like a single graph from `GMS`’s point of view. The federation router is the component that delivers that union: it fans queries out to multiple adapters, merges the results under a clear dedup rule, enforces per-adapter timeouts, and surfaces partial failures as data rather than swallowing them.

5.1 Router surface

The router is constructed against an `AdapterRegistry`—a small in-memory map of adapter name to `KnowledgeAdapter` instance, populated at application startup. The registry exposes `register(adapter)`, `unregister(name)`, `get(name)`, `all_adapters()`, `get_writable()`, and `get_with_capability(predicate)`—enough surface for the router to look up adapters by name, iterate them in registration order (which determines first-wins dedup tie-break, §5.2), and pre-filter on capability before fan-out.

The router presents the same operations as a single adapter, plus an optional `adapters` filter narrowing the fan-out (`None` = all registered). Reads fan out; writes are *triple-directed*: each `KALTriple.adapter` field names its destination, falling back to the first writable adapter only when `None`. The router never silently broadcasts a write across all stores—broadcasting would land the same logical fact in multiple backends with different identifiers, making federated reads return the same triple multiple times with no way to distinguish redundant from independent evidence.

```

class FederationRouter:
    def __init__(self, registry: AdapterRegistry, *,
                 conflict_strategy: ConflictStrategy =
                     ConflictStrategy.ALL,
                 default_timeout_ms: int = 5000) -> None: ...

    async def query_triples(self, query: KALQuery,
                           tenant_id: str | None = None,
                           adapters: list[str] | None = None,
                           ) -> KALQueryResult: ...

    async def search_similar_nodes(self, query_vector: list[float],
                                   tenant_id: str | None = None,
                                   adapters: list[str] | None = None,
                                   limit: int = 10,
                                   ) -> list[tuple[KALNode, float]]:
        ...

    async def insert_triples(self, triples: list[KALTriple],
                             tenant_id: str | None = None) -> int: ...

```

Figure 2 sketches the runtime topology. A single KALQuery from a GMS caller fans out through the router to capability-eligible adapters; each adapter consults its own backing store; results merge under content-hash dedup with per-adapter errors surfaced separately.

5.2 Fan-out and deduplication

`query_triples` dispatches concurrent calls to the selected adapters via `asyncio.gather(..., return_exceptions=True)`, then merges the results. Two design choices shape the merge.

Capability filtering at the router for gated operations. `search_similar_nodes` drops adapters lacking `supports_vector_search`; `insert_triples` routes only to `can_write` adapters; `query_triples` dispatches to every registered adapter (since `can_read=True` is the default and reads are not sub-divided). A defensive in-adapter check remains for both paths—an `AdapterError(error_type="capability")` indicates a router bug or a misreporting adapter.

Deduplication is by content_hash, not by external ID. Two adapters returning the same logical triple use different external identifiers by definition; federation dedups on the backend-independent `KALTriple.content_hash` (§4.3). The `KALTriple.adapter` field records the *successful origin* on the deduplicated row—only one value, even when several adapters contributed. Per-source attribution requires inspecting `KALQueryResult.errors` to distinguish “adapter B silent” from “adapter B timed out.”

5.3 Partial failure

Federated queries are inherently partial. The router’s contract is that **partial failure is normal**: a failing adapter does not abort the query and does not raise—it appends an `AdapterError` to `KALQueryResult.errors`, and other adapters continue. The classification table:

The caller contract: `KALQueryResult.errors` non-empty $\Rightarrow$ the result is *degraded* and must not be treated as a complete view. Partial visibility is more useful to GMS than no visibility, provided the partial-ness is visible to the caller.

¹`circuit_open` is reserved in the `AdapterErrorType Literal` but is not emitted in v1; callers matching on `error_type` should include it to stay forward-compatible. Separately, `AdapterError.message` is truncated to 200 characters before entering the result envelope (full text goes to the application log via `logger.exception`)—a defence against credential leakage, since adapter exceptions often embed connection URLs with passwords.

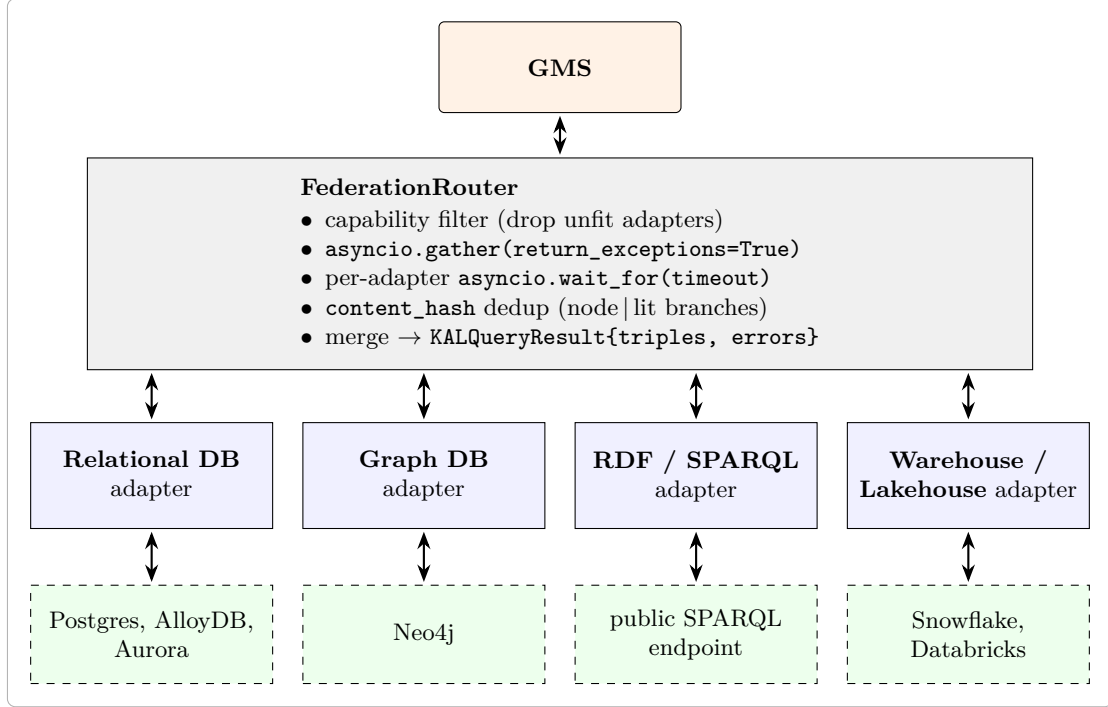


Figure 2: KAL federation topology with four illustrative adapter categories—**relational DB** (Postgres is the reference adapter; AlloyDB on GCP and Aurora on AWS are managed-cloud Postgres-compatible equivalents), **graph DB** (Neo4j as one example), **RDF / SPARQL endpoint**, and **analytics warehouse / lakehouse** (Snowflake, Databricks). The category labels in the adapter tier are deliberate: KAL’s protocol surface is store-shape-agnostic, so each box stands for a *family* of conformant backends rather than one specific vendor. Arrows are bidirectional throughout: queries flow down (GMS → router → adapter → backend), triples flow up; the router merges per-adapter results into a single `KALQueryResult`. Each adapter declares its own `AdapterCapabilities`; the router routes only to adapters that can serve a given operation. Partial-failure errors carry alongside successful triples in the merged result.

Writes are asymmetric. `insert_triples` does not silently degrade; per-triple failures raise `AdapterWriteError` carrying `adapter_errors: list[AdapterError]` (one row per failing adapter, same classification as reads) and `successful_count: int`. A silent degraded write leaves the graph inconsistent; a silent degraded read is recoverable. The protocol’s consistency cost falls on writes.

5.4 Timeout enforcement

Per-adapter timeouts are enforced via `asyncio.wait_for` with effective bound $t_{\text{eff}} = (\text{KALQuery.timeout_ms or default_timeout_ms})/1000$ seconds (per-call takes precedence; router default 5 s). The Python `or` treats 0 and `None` as the same signal (“use the configured default”)—a literal zero is almost always a caller bug; negative values are rejected at validation (`Field(ge=0)`). Each adapter’s `wait_for` runs concurrently, so slow adapters do not block fast ones; a trip emits `AdapterError("timeout")` while others continue. User-facing federation latency is $\max(\text{per-adapter latency})$ capped by the budget. Direct adapter calls (no router) must honour `KALQuery.timeout_ms` when set.

5.5 Latency budget

The 5-second default budget reflects the operating range of the adapter classes the protocol is designed to support (Table 1).

Adapters wrapping LLM-backed retrieval tools can blow past this budget because of an in-path parsing call. The protocol does not ban them but requires the cost to be visible at

Underlying condition	error_type
Per-adapter <code>asyncio.wait_for</code> timeout trip	"timeout"
Adapter raises an auth-related <code>KALError</code> subclass	"auth"
Transport raises <code>ConnectionError</code> / <code>OSError</code>	"connection"
Adapter cannot serve the operation (defensive check)	"capability"
Adapter skipped by a circuit breaker ¹	"circuit_open"
Any other exception	"internal"

Adapter class	Expected p50	Expected p99
Postgres (local)	5–20 ms	50 ms
Postgres (cross-region)	30–100 ms	200 ms
Cloud managed relational (AlloyDB, Aurora)	10–50 ms	200 ms
Neo4j	20–80 ms	300 ms
SPARQL endpoint	100–500 ms	5+ s
Analytics warehouse / lakehouse (Snowflake, Databricks)	200 ms–2 s	10+ s

Table 1: Per-adapter latency tiers as design targets. Budget rationale, not measured results—empirical characterisation is reserved for future work.

the boundary: expose the per-call `timeout_ms` knob and document the mitigation strategy (response cache, smaller parser model, batch pre-extraction) so production timeouts attribute to the responsible adapter rather than to “the federation layer.” Analytics-warehouse adapters face the same budget problem from the storage side—cold-cache scans and large per-query compute mean direct query-path use rarely fits an interactive budget—and the natural mitigation is the §7.3 Pattern 3 default: extract once upstream, persist into a Postgres-tier store, expose that store through KAL. Callers should avoid chaining federated reads in tight loops—each pays the slowest-adapter tax.

5.6 Conflict resolution

When deduplication collapses two adapter responses that share a `content_hash` but disagree on metadata, the router applies a `ConflictStrategy`: `ALL` (default—keep the first-registered adapter’s copy; no metadata-based re-ranking), `HIGHEST_CONFIDENCE` (keep the largest `verification.confidence`), or `MOST_RECENT` (keep the latest `provenance.extracted_at`). `ALL` is the default because registration order is deterministic across runs and makes no assumption that metadata is comparable across heterogeneous adapters; per-source attribution remains visible on `KALTriple.adapter` and through `KALQueryResult.errors`. The metadata-aware strategies serve callers that have a principled cross-adapter comparison and want to override registry order.

The three strategies resolve horizontal-axis conflicts. On the vertical axis of §3.5, `HIGHEST_CONFIDENCE` and `MOST_RECENT` are meaningful only when the participating verifiers share a calibration; for independently-trained verifiers `ALL` is the only safe default, returning one row per `(content_hash, verifier)` and leaving cross-verifier resolution to the caller. A `PER_VERIFIER(name)` strategy is reserved for v2 (§9).

5.7 What federation does not promise

Three explicit non-promises, all of which apply to both axes: federation does not synthesise triples (the protocol is a data-plane spec, not an inference engine), does not guarantee stable read order across calls (callers needing determinism sort the result), and does not implement cross-adapter transactions (reads are eventually consistent; writes route per-triple). Vertical-axis-specific non-promises mirror these: federation does not synthesise verdicts (one verifier’s verdict is never forged for another), does not guarantee stable verdict ordering across calls, and does not implement cross-verifier transactions (a parent agent’s roll-up of sub-agent verdicts is

the caller’s responsibility, not the router’s).

5.8 Federation along the GMS axis

The vertical axis of §3.5 uses the same router, dedup primitive, and partial-failure contract as the horizontal. Four mechanics differ:

Per-verifier read filtering. A `verifiers: list[str] | None = None` filter on `KALQuery` mirrors the horizontal `adapters` filter; `None` means all verifiers. Reserved for v2—v1 callers filter in-process.

No vertical deduplication. Verifiers are never collapsed: federation returns one row per `(content_hash, verifier)` pair under `ConflictStrategy.ALL`, and the caller composes verdicts.

Partial-failure semantics carry over from §5.3. A silent verifier surfaces as an `AdapterError` in `KALQueryResult.errors` with the verifier (or its proxy) named in the `adapter` field; non-empty `errors` means the multi-verifier view is degraded.

Writes are per-verifier. `update_verification` is the vertical-axis write path; each call’s `VerificationMetadata.verifier` names the authoring GMS, and the adapter persists one row per `(triple, verifier)` without overwriting other verifiers’ verdicts. The §5.3 write asymmetry carries unchanged.

6 Schema Discipline for GMS-Compatible Knowledge Graphs

GMS’s 100% verification accuracy in [Sudjianto et al. \[2026c\]](#) holds *given* a well-formed input graph; ill-formed graphs produce ill-defined verdicts. The Knowlytix ingestion pipeline applies the discipline as it builds the graph; customer graphs built independently must satisfy it explicitly. The discipline organises into four incremental layers—Layer 1 (entity identity) for read-only single-store use, Layers 1–3 for full GMS verification, Layer 4 only when the customer operates more than one store.

6.1 Layer 1 — Entity identity

Every entity must be **typed**, **stable**, and **normalisable**.

Typed. Each entity carries one or more type tags from a controlled vocabulary (`KALNode.node_types`). GMS consumes the type at two points: embedding lookup, where untyped entities cannot be placed in GMS’s space with a meaningful neighbourhood; and contradiction comparison, where mixing types produces meaningless outputs. Customers with untyped graphs must add a type-tagging pass or accept coarse-grained GMS output.

Stable. `KALNode.id` must not change across queries within a session—GMS’s runtime integer index fragments under ID churn, making verdicts order-dependent. Stable need not mean immutable: backends can represent entity merges however their storage supports (the reference Postgres schema uses a `merged_into` self-reference), but adapters must either return the survivor transparently or refuse the query, never leak stale identifiers. Exposing merge state at the protocol boundary (`KALNode.merged_into` or `MergedNodeError`) is reserved for a later revision.

Normalisable. Each entity carries a `normalized_name` derived deterministically from the display name (lowercase, whitespace collapsed, acronyms expanded). The normalised form is the dedup key inside a single adapter and across federated adapters. Backends that cannot compute

it server-side can defer to the adapter implementation; what is unacceptable is two adapters normalising differently for the same input.

6.2 Layer 2 — Triple semantics

A triple is an *assertion* $\langle s, p, o \rangle$ where s is a `KALNode`, p is a relation from a typed vocabulary, and o is either a `KALNode` or a `KALLiteral`. Two choices distinguish GMS-compatible triples from generic RDF.

Predicate vocabulary is controlled. Predicates are stored explicitly with synonyms registered against canonical forms—stricter than open RDF, where any URI can be a predicate. The reason is GMS’s path-consistency check, which requires every relation in the graph to have a corresponding registered transform on the GMS side. Unregistered predicates produce undefined outputs and silently break path-consistency reasoning.

Edge properties are separated from node metadata. Provenance, confidence, and verification metadata attach to the *assertion*, not to its entities. `KALTriple.properties` carries edge metadata (qualifiers, weights); the nested `KALTriple.verification` carries the first-class verification fields (`confidence`, `verification_status`, `scores`, `verifier`); `KALTriple.provenance` carries source attribution. This separation lets the same entity participate in many assertions at different confidence levels without polluting the entity record. Backends that conflate edge and node properties (some property-graph stores) can adapt, but `update_verification` must avoid race conditions when multiple triples share a node.

6.3 Layer 3 — Literal handling

KAL represents value-typed objects as `KALLiteral` with `value` (lexical form), `datatype` (XSD-style), and `language` (BCP 47). Two design points matter.

Datatype is load-bearing. GMS compares numeric quantities by value, not by lexical form; the strings `"$394.3 billion"` and `"394300000000"` are equal under `xsd:decimal` and unequal under `xsd:string`. Backends storing literals as untyped strings can adapt, but datatype assignment must happen at conversion time and must be deterministic.

The dedup key extends naturally. `content_hash` (§5.2) has two branches: $(s_{\text{norm}}, p, \text{"node"}, o_{\text{norm}})$ for node objects and $(s_{\text{norm}}, p, \text{"lit"}, v, d, \ell)$ for literal objects. The discriminator prevents collisions between a node named `"$394.3B"` and a literal value `"$394.3B"`—a real pathology in finance and clinical KGs where numerics sometimes appear as identifiers.

6.4 Layer 4 — Federation identity

When a customer operates more than one store, the same logical entity will appear in each store under a different identifier. Consider the entity *Foxglove BioSciences, Inc.* in a deployment with three connected backends. In Postgres it might live as a row keyed by UUID `a1b2c3d4-...`; in Neo4j as a node with internal integer ID `4821`; in a SPARQL endpoint as the URI `http://example.org/company/FOXG`. Each identifier is correct for its own store. None of them, on its own, identifies the entity *across* stores. Two structures handle this without conflating the two roles those identifiers serve.

The external-ID registry: a translation table for backend-native handles. The registry is a mapping $(\text{tenant_id}, \text{source_adapter}, \text{external_id}, \text{resource_type}) \rightarrow \text{internal_id}$ that, for every external identifier any adapter has observed, records the canonical internal UUID it corresponds to. When the Neo4j adapter sees node `4821`, it consults the registry to find the

same internal UUID that the Postgres adapter resolves for `a1b2c3d4-...`, so the platform’s internal representation of *Foxglove BioSciences, Inc.* is the same entity regardless of which adapter surfaced it. The composite uniqueness constraint (all four columns together) prevents corruption when an adapter retries the same ingestion—two retries of the same observation collapse into one registry row rather than spawning two competing internal UUIDs.

Dedup is by normalised name, not by external ID. When two adapters return what is logically the same entity, their external IDs differ by definition—a UUID and a node integer cannot match—so external ID is useless as a dedup key. Federation instead dedups on the normalised name from §6.1: two entities with the same `normalized_name` (`"apple inc"` in our example) are the same entity for dedup purposes, regardless of which IDs their source adapters minted. The registry is the *write-back* path, not the *dedup* path: dedup runs on names so federated reads coalesce cleanly; the registry runs on IDs so a GMS verdict written to one adapter can be propagated to the same entity in another. Conflating the two—using external ID as the dedup key—is the most common customer-integration mistake. It produces a federation that *appears* to work (each adapter returns its own triples) but silently emits duplicate triples for every entity that exists in more than one store.

Layer 4 requires a federation metadata store. The registry needs a transactional relational store. In the reference deployment it co-locates with the primary Postgres KG (sibling `knowlytix.external_id_registry` table, same instance) so the registry row commits in the same transaction as the entity it points to. Customers whose primary KG is Neo4j, SPARQL, a managed cloud relational store (AlloyDB, Aurora), or an analytics warehouse (Snowflake, Databricks) can host the registry in a small separate Postgres/MySQL instance; atomicity reduces to write-ahead-then-reconcile (idempotent `ON CONFLICT DO NOTHING` makes registry rows safe to re-insert on retry). The two configurations differ in atomicity guarantee, not in contract—the router consumes the four-column composite key and forward/reverse lookups regardless of storage location.

Departmental scope within an organisation. For organisations with internal departments—the bank deployment of §3.5—each department maps to its own `tenant_id`, with cross-department reads mediated by an application-layer read-lease. A typed `scope_id` field is a v2 candidate (§9).

Figure 3 sketches the identifier flow: an external identifier enters through an adapter, resolves against the registry to an internal UUID, and surfaces in GMS as a runtime integer index (used for fast lookup inside the geometric state). Writes are one-way inward: ingestion creates internal entities via `find_or_create`, and autotuning (§8) lands GMS-proposed mutations on the internal KAL triple KG through the same protocol surface. External KGs are never written to.

6.5 Two domain examples

The four-layer discipline carries across domains; only the vocabulary changes.

Finance (FinStructBench-aligned). *Entities:* companies, business units, financial metrics, time periods, regulatory categories. *Predicates:* `has_revenue`, `has_capital_ratio`, `subsidiary_of`, `reports_under`, `effective_during`. *Literals:* monetary values as `xsd:decimal` with currency/unit qualifiers on the edge; dates as `xsd:date`. *Registry mappings:* internal UUIDs ↔ SEC CIK numbers ↔ EDGAR filing identifiers. The schema of Sudjianto [2026a].

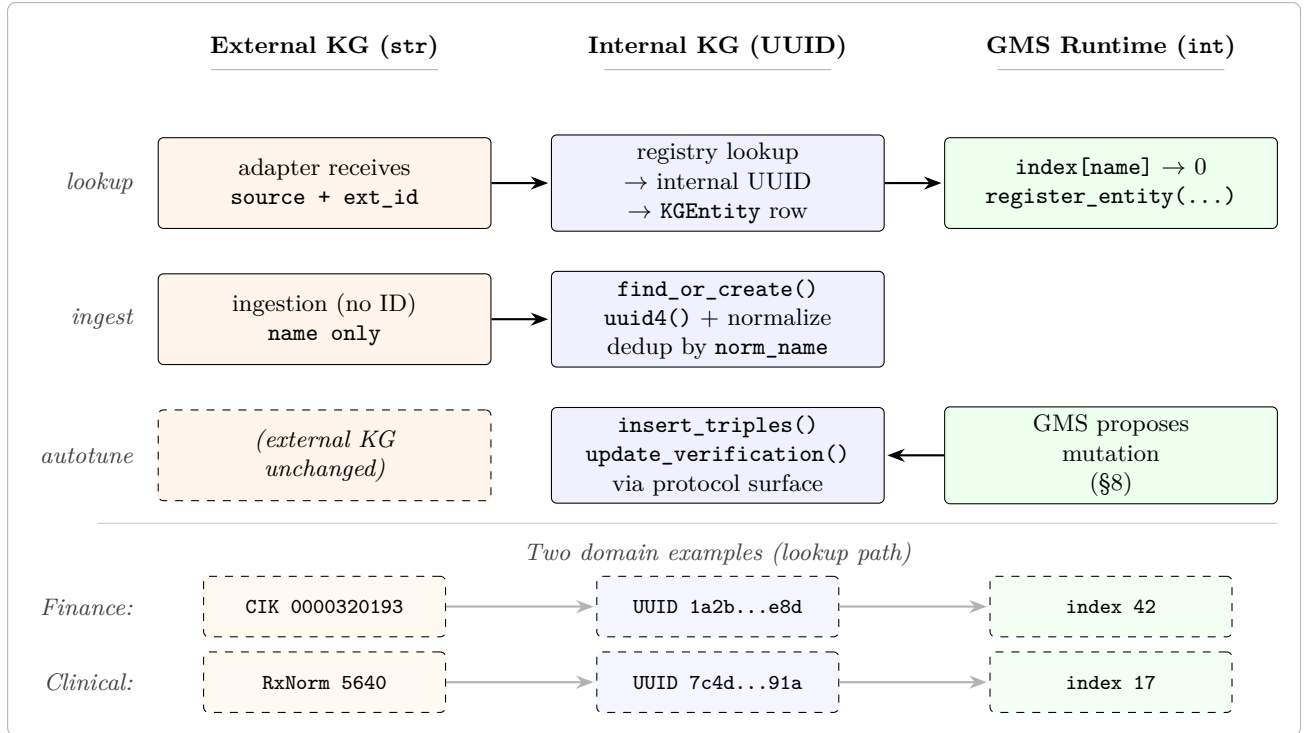


Figure 3: ID flow across the external / internal / runtime boundaries. The top three rows show the three operations that cross the boundary — *lookup* (an external ID resolves to an internal UUID and a runtime index), *ingest* (a name-only entity is assigned a fresh UUID via `find_or_create`), and *autotune* (GMS-proposed mutations land on the internal KAL triple KG through the same protocol surface; external KGs are never written to). The bottom two rows show the same *lookup* path realised in two domains: SEC CIK numbers (finance) and RxNorm CUIs (clinical) round-trip through the same registry to UUIDs and runtime indices, illustrating that the boundary structure is identifier-agnostic.

Clinical decision support. *Entities:* drugs, conditions, populations, contraindication classes. *Predicates:* `contraindicated_with`, `metabolised_by`, `dose_range_for`, `interacts_with`. *Literals:* dose values as `xsd:decimal` with mg / mg/kg unit qualifiers; half-life as `xsd:duration`; MeSH codes as opaque strings. *Registry mappings:* internal UUIDs ↔ RxNorm CUIs ↔ SNOMED CT ↔ DrugBank SPARQL URIs. The predicate vocabulary is typically denser (`weakly_inhibits`, `strongly_induces`, etc.); each must be registered against a GMS transform before path-consistency checks can use it.

6.6 What the discipline rules out

Most production knowledge graphs can be adapted to the discipline without rebuilding, but three categories cannot serve as primary GMS substrates without modification:

- **Bag-of-strings retrieval indices** (vector stores, BM25 corpora over unlabelled text) lack typed entities for embedding lookup. The protocol allows them to participate as `search_similar_nodes` providers only, with the consumer type-validating results—a forward-looking property, not a v1 integration.
- **Property graphs without a controlled predicate vocabulary** lose the path-consistency gate (no registered transforms); plausibility and contradiction gates still function.
- **Stores without per-triple provenance and verification fields** can supply candidate triples but `update_verification` becomes meaningless—retrieval-only participants in the verification loop.

What the discipline does *not* rule out: heterogeneous storage (relational + native graph + RDF in one deployment), heterogeneous identifier formats, varying query capabilities, or partial coverage. The layering is deliberate—an adapter satisfying Layers 1–2 only is still useful for non-numeric single-store verification.

7 Reference Adapter Design

To ground the specification, we describe one reference adapter: `PostgresKnowledgeAdapter`, sitting over the Knowlytix platform’s existing relational schema. Postgres anchors the reference because the platform owns the storage (dedup, foreign-key, and tenant-isolation rules are already production-hardened, so the adapter delegates rather than reimplements), it exercises every capability KAL declares, and its latency anchors the low end of the \$5.5 budget. Adapters over Neo4j, SPARQL, and third-party-tool transports follow the same pattern—capabilities declared explicitly, GMS metadata in typed fields, tenancy at the boundary, identity through the registry—with backend-specific constraints (synthesised provenance, initial `unverified` status, per-call latency mitigations) deferred to future work.

7.1 PostgresKnowledgeAdapter

Four design choices anchor the Postgres adapter.

Wrap, do not rewrite. Every KAL operation delegates to an existing storage-layer function—repository helpers for reads, the service-layer claim-persistence path for `insert_triples` (Table 2). The adapter is a thin protocol facade, not a parallel implementation; reimplementing any production-hardened invariant would diverge within weeks of the first downstream change.

KAL operation	Storage backing
<code>query_triples</code>	Existing triple-fetch helpers in <code>kg.repository</code> ; KALQuery patterns translated to SQL
<code>get_node</code>	Existing single-entity fetch in <code>kg.repository</code>
<code>query_adjacent_triples</code>	Existing adjacency query plus direction filtering in adapter
<code>insert_triples</code>	<code>kg.service.persist_claims_to_kg</code> (the service-layer claim-persistence path with built-in dedup; reused intact from the existing ingestion pipeline)
<code>update_verification</code>	Direct ORM update against typed columns on the triples table
<code>delete_triples</code>	Existing cascade-aware delete in <code>kg.repository</code>
<code>search_similar_nodes</code>	pgvector $\leftrightarrow$ cosine-distance query
Text search via <code>KALQuery.text_search</code>	tsvector / tsquery against a GIN index on entity names

Table 2: `PostgresKnowledgeAdapter` operation mapping.

GMS metadata in typed columns, not JSONB. Verdict fields (`confidence`, `verification_status`, `verified_at`, `verifier`, plus `scores` JSONB for per-engine sub-scores) live in typed columns on the triples table. Mechanical reason: `KALQuery.min_confidence` pushes down into the SQL plan as an indexed predicate, impossible inside a JSONB document. Positional reason: this metadata is the platform’s differentiator from generic retrieval; opaque blob storage demotes it precisely where it matters most.

Map KALLiteral to existing partial-literal columns. The triples table carries `object_type`, `object_numeric`, and `object_numeric_hash`. Node-object triples set `object_id` / `object_type="entity"`; numeric literals set `object_type="numeric"` and `object_numeric`. Non-numeric literals are currently coerced to node-typed objects (lossy—`datatype` and `language`

dropped) pending a follow-up migration adding dedicated `object_datatype` and `object_string` columns. We surface this design debt deliberately: perfect round-trip is sometimes a follow-up migration away on non-greenfield schemas.

Capabilities declared loud, tenancy enforced at the boundary. The Postgres adapter advertises every capability (`can_read`, `can_write`, `can_delete`, `supports_vector_search`, `supports_text_search`, `supports_provenance`, `supports_verification_metadata`, `supports_atomic_writes`) because the storage actually backs every claim; restricted backends should narrow rather than overclaim. Tenancy is non-negotiable—the adapter raises `TenantRequiredError` when called without a `tenant_id`, with no fallback. One adapter quietly skipping the tenant filter loses isolation across the whole federation.

7.2 What the reference design illustrates

The four design choices walk through four dimensions any customer adapter author faces:

- **Identity**—local writes resolve to canonical internal UUIDs; federated writes naming a foreign `KALTriple.adapter` additionally register their external IDs in `knowlytix.external_id_registry`. Read-path registry consumption is deferred; normalised-name matching handles dedup today.
- **Provenance**—typed columns rather than JSONB blob, queryable through ordinary SQL.
- **Verification metadata**—read/written via `update_verification` with `min_confidence` pushed into the SQL plan via a dedicated index—the GMS-vs-generic-retriever gate is index-resident.
- **Latency**—fast and predictable, anchoring the low end of the §5.5 budget; LLM-in-the-loop tails surface in adapters yet to come.

Adapter authors targeting Neo4j, SPARQL, or third-party-tool transports vary these dimensions along the same axes; the §6 discipline holds across them.

7.3 Adapter patterns for non-triple-ready sources

§3.4 assumes the KG is already triple-shaped, but many real sources are not—operational relational tables, property graphs without a controlled predicate vocabulary, vector stores of unlabelled chunks, third-party retrieval APIs returning text. KAL embraces these rather than turning them away: the protocol boundary stays at `KALTriple`; the adapter is where triple-shaping happens. Three patterns cover the realistic cases.

Pattern 1: Schema projection (the cheap case). Structured-but-not-triple sources project into triples through a deterministic mapping. A relational row’s primary key is the subject; a foreign-key column is a predicate; the referenced row is the object; literal columns become $(s, p, \text{literal})$ triples. Property-graph nodes map similarly. R2RML formalises this for RDF; the idea generalises to any structured source. No LLM enters the query path; latency stays at the Postgres-tier budget. Right pattern for an existing operational database.

Pattern 2: Query-time synthesis (the LLM-tail case). Unstructured-but-retrievable sources (vector stores of chunks, MCP tools returning text, foundation-model search APIs) can only be triple-shaped on demand. The adapter receives a `KALQuery`, issues a backend retrieval, runs an extraction step (LLM call, structured-output prompt, rule-based parser), and emits `KALTriples`. The per-call `timeout_ms` knob is the budget control; the implementation exposes its mitigation strategy (response cache, smaller parser, batch pre-extraction) so production timeouts attribute correctly. General but pays per call—appropriate when the source is live and pre-processing is impractical.

Pattern 3: Pre-extraction upstream (the recommended default). Run extraction once (subject of the forthcoming extraction paper), persist triples in a primary store, expose that store through KAL. Separates slow LLM-driven extraction from fast deterministic retrieval; the §6 discipline is satisfied at extraction time, not at query time; the adapter reduces to Pattern 1 over the extracted store. For data that does not change in real time—most of it—this is the recommended default.

Composition under federation. A realistic deployment mixes all three. A finance customer’s pre-extracted obligation graph (Pattern 3, Postgres adapter) sits alongside their transactional general ledger (Pattern 1, schema-projected from a different Postgres schema) and their live regulatory news feed (Pattern 2, MCP adapter with an LLM in the query path). The federation router (§5) fans queries out concurrently; each adapter contributes triples scoped to what its pattern allows; the partial-failure semantics of §5.3 keep slow Pattern-2 sources from blocking fast Pattern-1 ones. The caller sees one triple stream and one error envelope. This composition is the structural reason KAL is worth implementing in the first place: a single protocol surface accommodates the full heterogeneity of real customer data without requiring any single source to fit a uniform shape.

7.4 Connection lifecycle and credential management

The protocol surface in §4 deliberately stops at *what an adapter does*—it omits *how the adapter got connected in the first place*. In a multi-tenant deployment with adapters added and revoked over time, this is its own layer of design, and the reference implementation places it strictly *above* `KnowledgeAdapter` rather than inside it. An adapter registry holds one connection record per (`tenant_id`, `adapter_name`), populated either at startup from configuration or at runtime through a management API that can add, validate, disable, or remove adapter connections without an application restart and without code changes. Credentials live in the registry encrypted under a rotatable key—key rotation re-encrypts existing records without re-entry of underlying credentials, so a key compromise is a tractable operational event rather than a per-tenant re-onboarding. A handshake routine at registration time and on a periodic schedule thereafter checks connection liveness and capability consistency (the declared `KnowledgeAdapterCapabilities` still match what the backend reports), surfacing drift as a registry-level health signal rather than as a runtime `AdapterError`. Two operational invariants are non-negotiable: raw backend responses and credential material are never written to logs (the `AdapterError.message` truncation in §5.3 is the protocol-level half of the same discipline), and the management API itself runs above the federation router so that lifecycle operations cannot be smuggled into a `query_triples` call.

Keeping connection lifecycle *out* of the protocol is the load-bearing choice. It means the protocol stays the minimum surface a backend implementer has to fill in—capabilities, the eight operation methods, the value envelope—and adapter authors are not asked to invent their own rotation schemes, handshake conventions, or credential-storage policies. A connection-lifecycle subsystem then composes orthogonally: bundled with `knowlytix-kal` for self-hosted deployments, supplied by the host platform in a managed deployment, or—for an organisation that already has a secrets manager and a service registry—wired into existing infrastructure without changing a single line of adapter code.

8 KG Autotuning: A Self-Improving Consumer Mode

The other consumer modes covered in §3.1—verifier, agent memory, validation harness—treat the underlying knowledge graph as fixed input. *Autotuning* closes the loop by feeding geometric signals back into proposals to mutate the KG. It is the mode that makes KAL-readiness *durable* under drift rather than merely true at the moment of ingestion. Full algorithmic detail is the subject of a separate paper in this series; this section sketches the consumer-mode contract autotuning places on KAL.

8.1 The autotuning loop

Autotuning observes geometric signals that GMS already computes, ranks them into typed proposals, gates each proposal, and applies approved mutations to the KG. The current implementation productionises eight signals (Table 3).

Signal	Source	Mutation it suggests
Anomalous triple	Geodesic distance z-score	<code>delete_triples</code>
Missing triple	Link prediction via cosine similarity	<code>insert_triples</code>
Redundant relation	Frobenius distance between rotation transforms	Merge predicates onto canonical form
Duplicate entity	Cosine similarity between embeddings	Merge entities (canonical $\rightarrow$ alias)
Phase encoder drift	Numeric encoder range calibration	Recalibrate without KG mutation
Low-quality entity	Name-format anomalies + embedding outliers	<code>update_verification</code> (downgrade confidence)
Extraction mismatch	Recurring fuzzy resolutions	Add learned alias
ENM key misalignment	Orphan keys vs. unreachable claims	Add alias or flag missing entity

Table 3: Autotuning signals, the GMS-side computation that produces them, and the KAL mutation each one proposes. Eight signals are productive in the current implementation; the protocol surface they consume is the same one §4 already defines.

Each signal produces a typed proposal with a confidence score. Proposals route through a two-tier gate: high-confidence (**AUTO**) commits immediately under rate limits; lower-confidence (**REVIEW**) accumulates for human approval. Every applied mutation writes an audit row supporting rollback. The cumulative effect is a KG that gets demonstrably better with use—vocabulary mismatches become exact aliases, redundant relations collapse onto canonical forms, surviving anomalous triples get flagged. This is a class of improvement no model retraining could produce: the corrections are graph-shaped, not prompt-shaped.

8.2 What autotuning needs from KAL

Autotuning’s data-plane contract is exactly the §4 surface—no protocol extension (Table 4).

Autotuning requirement	KAL surface
Compute signals over the current graph state	<code>query_triples</code> , <code>query_adjacent_triples</code>
Record verdicts that downgrade or confirm a triple	<code>update_verification</code>
Apply approved additions (aliases, predicted-missing triples)	<code>insert_triples</code>
Apply approved deletions (anomalous triples)	<code>delete_triples</code>
Signals over entity / predicate vectors	Implicit in the §6 schema discipline

Table 4: Autotuning’s requirements map one-to-one onto the existing §4 surface.

The §6 schema discipline is what makes autotuning possible at all. Frobenius distance between relation transforms requires registered predicates; cosine outlier signals for entities require typed entities; link prediction requires stable identity across batches. Customers who satisfy §6 Layers 1–3 unlock both verification and autotuning; Layer 1 alone unlocks plausibility scoring but not the path-consistency signals that higher-confidence mutations depend on. **The discipline pays for itself twice.**

An adapter advertising `can_write=False` participates as a *signal source only*—autotuning reads it for signal computation, mutations route to writable source-of-truth adapters. The same federation router (§5) fans signal-collection queries with identical partial-failure semantics, so a slow adapter contributes nothing to that cycle but does not halt it.

8.3 Federation: autotuning across heterogeneous backends

Multi-adapter autotuning reuses three §5 properties without modification:

- **Concurrent fan-out.** Signal computation dispatches via the same `asyncio.gather` path; partial failure surfaces as `KALQueryResult.errors`.
- **Per-triple write routing.** Autotuning mutations carry the originating adapter—`KALTriple.adapter` names the destination, so the router writes back to where the data came from.
- **Capability and verifier gating.** `can_write=False` adapters participate only in signal collection. When multiple GMS instances autotune over a shared adapter (§3.5 vertical axis), each proposes mutations under its own `verifier`; conflicting proposals accumulate in `REVIEW`, with the §8.1 human-gating layer disambiguating.

The result is a single self-improving loop over heterogeneous backends—Postgres claim store, SPARQL obligation graph, third-party vendor catalog all contributing scoped signals in one cycle.

8.4 What this section establishes

Autotuning is not part of the KAL protocol but a consumer mode it enables. Three claims follow:

1. **KAL is the right boundary.** The same value envelope that supports verification supports the autotuning feedback loop without protocol extension—the strong form of §3.3’s separation-of-concerns argument.
2. **The schema discipline is twice-paying.** The same Layer 1–3 properties that unlock GMS verification unlock the autotuning signals built on top of it.
3. **GMS lands as a solution, not just an algorithm.** Verification *plus* autotuning *over* KAL is a self-improving knowledge system: the KG gets demonstrably better with use, no model retraining and no schema migration required. The same loop composes across multiple GMS instances sharing a backend (§3.5).

How fast autotuning improves a KG in deployment—mutation review rates, cycle cadence, per-domain gating thresholds—is reserved for the autotuning paper.

9 Discussion and Limitations

9.1 What KAL is not

Coverage-bounded. GMS reasons against assertions that exist in at least one connected adapter. A claim no adapter holds is flagged *not in scope*, not *false*. Whatever consumer mode is calling—verifier, agent memory, harness—must handle the “I cannot tell” branch itself.

Not a hallucination eliminator on its own. GMS verifies *structured triples*; an LLM producing free-form text has not yet been reduced to the typed-triple form GMS consumes through KAL. Extraction errors upstream—wrong subject, missed entity, hallucinated predicate—are outside KAL’s remit. The design of the extraction layer that produces GMS-compatible triples from source documents is the subject of a separate paper in this series, currently in preparation. The chain is only as strong as its weakest link.

No query planning. The protocol accepts AND-only patterns; there is no join optimisation, source selection beyond capability filtering, or cardinality estimation. Customers can implement an optimiser inside an adapter (a SPARQL adapter can wrap a federated SPARQL planner), but the federation router itself does not.

No cross-adapter transactions. Writes route per-triple; no rollback spans adapters. The distributed-transaction story across SPARQL, Neo4j, and relational stores does not have a clean general answer, and the protocol’s contract is intentionally minimal here.

9.2 What KAL pushes onto the customer

The schema discipline of §6 is a contract the customer’s KG must satisfy.

Typed entities and registered predicates are non-negotiable for full GMS consumption—particularly the path-consistency check. In fast-moving regulatory domains a predicate-registration workflow is needed to keep GMS’s relation registry in step.

Identity stability must be the customer’s commitment, not something the adapter papers over. The `external_id_registry` helps only if external IDs are themselves stable enough to be useful registry keys.

Mitigation choices for slow adapters (cache TTL, parser model, batch pre-extraction, aggressive timeouts) are deployment-specific—the protocol cannot pick a default that suits both interactive flows and offline pipelines.

9.3 Empirical questions deferred

This is a design paper. Several deployment questions are deliberately reserved for a follow-up:

- **Abstraction integrity under backend swap**—do GMS verdicts agree when the same logical KG is served by Postgres, Neo4j, or another conformant backend in turn?
- **Federation behaviour under partial failure**—at what adapter-failure rate does federation become operationally unreliable?
- **Latency in practice**—does the 5 s budget hold in real deployments, including for adapters that wrap LLM-backed retrieval tools?
- **Cross-domain schema generality**—does the discipline carry across a third and fourth domain at production scale?
- **Cross-verifier agreement on the vertical axis**—at what rate do multi-GMS deployments produce conflicting verdicts on the same triple, which categories of conflict are recoverable through the §5 ALL strategy versus those requiring human policy above the data plane, and how does the rate vary between sub-agent hierarchies (shared calibration) and bank-department deployments (independent calibration)?

Measurements require deployed adapters at scale, which will exist by the time the follow-up paper is plausible.

9.4 Open design questions

Schema introspection. §4.4 omits introspection methods. Works for verifiers that know which predicates to query; works less well for autonomous agents discovering an unfamiliar adapter. A future `list_predicates()` adds surface area and a non-trivial back-pressure question for large vocabularies.

Streaming and notifications. The protocol is request/response. A change-feed extension would let verifiers react to triple changes without polling, but many backend transports (especially third-party tool servers) cannot push, and the safe v1 default is to leave it out.

Formal sufficiency of the schema discipline. §6 claims Layers 1–4 are *sufficient* for GMS compatibility. The claim is informally argued and empirically tractable but not formally proven. A formal characterisation—in category-theoretic or typed-graph-signature terms—would distinguish necessary from incidental design constraints. We leave it as a clean open problem.

Typed verifier and scope identity. §4.3 carries verifier identity as a stable string; §6.4 carries department scope as a separate `tenant_id`. Both work in v1 and require no protocol extension. A future revision could promote `verifier` to a typed `VerifierIdentity` record (calibration root, version, capabilities) and add a typed `scope_id` field on the protocol—enabling `ConflictStrategy.PER_VERIFIER` and cross-department federation without an application-layer read-lease. The v1 string-plus-tenant convention is intentionally lightweight; production deployment data on multi-GMS deployments (the empirical bullet above) will tell us when the convention starts to chafe.

10 Conclusion

The three preceding papers develop GMS as an algorithm, as something measurable, and as a verification primitive in a multi-agent pipeline. None addresses how a knowledge graph should be shaped so that GMS has something to verify against. This paper closes that data-plane gap.

KAL has three design components: a typed protocol exposing any knowledge graph through a small operation surface and an explicit capability declaration; a federation router with content-hash deduplication, per-adapter timeouts, and explicit partial-failure semantics; and a four-layer schema discipline. A reference adapter design—`PostgresKnowledgeAdapter`—grounds the specification in code; adapters over Neo4j, SPARQL, managed cloud relational stores (AlloyDB, Aurora), analytics warehouses / lakehouses (Snowflake, Databricks), and third-party-tool transports follow the same pattern and are deferred to future work. The protocol is symmetric under two federation axes (§3.5), so sub-agent hierarchies and multi-department enterprise deployments fall out of the same spec without protocol extension.

KAL does not introduce new geometric machinery. Its contribution is the layer immediately below GMS: the contract a customer architect can satisfy without writing GMS code, the schema discipline that makes GMS output a property of *the design* rather than of *the implementation*, and the federation semantics that let GMS see a coherent view across heterogeneous storage. The autotuning consumer mode of §8 closes the loop—feeding the same geometric signals back through the same protocol surface to propose KG mutations under human gating—so KAL-readiness becomes durable under drift rather than a one-time onboarding contract, and GMS lands as a self-improving knowledge system rather than as a verifier-in-isolation.

The protocol module is released as `knowlytix-kal` (Apache-2.0), mirroring the FinStruct-Bench open-source pattern. Empirical evaluation across deployed customer backends is reserved for a follow-up. We expect the design to evolve under contact with production, and we expect to learn from contributors who try to satisfy the schema discipline against backends and domains we have not anticipated. The intended state, three releases from now, is that a customer architect can build a GMS-compatible knowledge graph on top of their existing storage by reading this paper and the reference adapter sources, in that order, and producing a working adapter in an afternoon.

References

- Acosta, M., Vidal, M.-E., Lampo, T., Castillo, J., & Ruckhaus, E. (2011). ANAPSID: An adaptive query processing engine for SPARQL endpoints. In *Proceedings of the 10th International Semantic Web Conference (ISWC)*, pp. 18–34. https://doi.org/10.1007/978-3-642-25073-6_2
- Anthropic. (2024). *Model Context Protocol specification*. <https://modelcontextprotocol.io/>
- Cognee Project. (2024). *Cognee: Open-source AI memory engine for LLM applications*. [Computer software]. <https://github.com/topoteretes/cognee>
- Cyганиак, R., Wood, D., & Lanthaler, M. (Eds.). (2014). *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation, 25 February 2014. <https://www.w3.org/TR/rdf11-concepts/>
- Görlitz, O., & Staab, S. (2011). SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In *Proceedings of the 2nd International Workshop on Consuming Linked Data (COLD)*, vol. 782. CEUR-WS.
- kg-gen Project. (2025). *kg-gen: Knowledge graph extraction with language models*. [Computer software, MIT licence]. <https://github.com/stuffyai/kg-gen>
- Lebo, T., Sahoo, S., & McGuinness, D. (Eds.). (2013). *PROV-O: The PROV Ontology*. W3C Recommendation, 30 April 2013. <https://www.w3.org/TR/prov-o/>
- Schwarte, A., Haase, P., Hose, K., Schenkel, R., & Schmidt, M. (2011). FedX: Optimization techniques for federated query processing on linked data. In *Proceedings of the 10th International Semantic Web Conference (ISWC)*, pp. 601–616. https://doi.org/10.1007/978-3-642-25073-6_38
- Sudjianto, A. (2026a). FinStructBench: A benchmark for structured information retrieval from financial documents using graph-verifiable questions. SSRN. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6506403 (*Knowlytix series paper 1.*)
- Sudjianto, A., & Lau, W. (2026b). When the judge is wrong: Measuring LLM-as-judge reliability against graph-verified ground truth in financial documents. SSRN. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6482162 (*Knowlytix series paper 2.*)
- Sudjianto, A., Nugroho, A., Lau, W., & Prabunath, P. (2026c). Policy-governed agentic AI for interpretable machine learning: GMS-driven multi-agent pipeline with deterministic verification and governed LLM reasoning. SSRN. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6567199 (*Knowlytix series paper 3.*)
- Taelman, R., Van Herwegen, J., Vander Sande, M., & Verborgh, R. (2018). Comunica: A modular SPARQL query engine for the web. In *Proceedings of the 17th International Semantic Web Conference (ISWC)*, pp. 239–255. https://doi.org/10.1007/978-3-030-00668-6_15