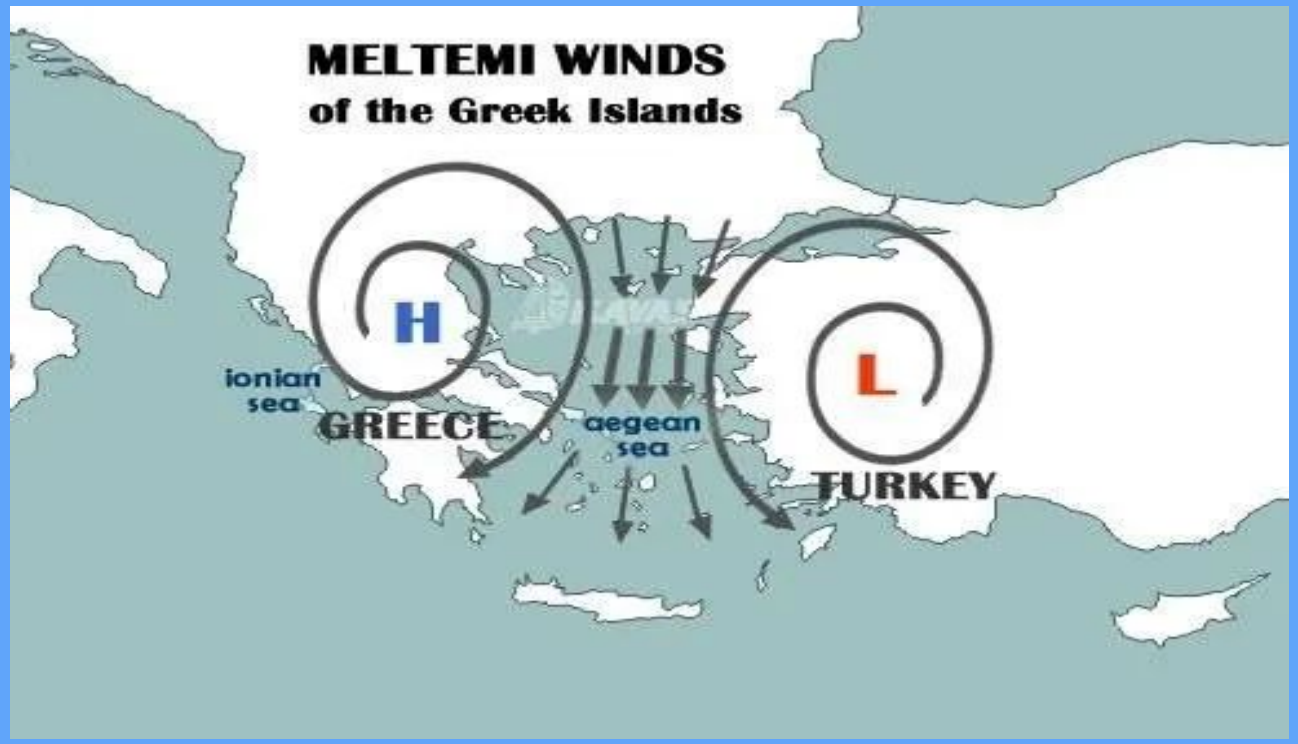
Bangkok, Thailand | March 11, 2026

Alkin Tezuysal
Director of Services at Altinity

PERCONA UNIVERSITY

Powering AI-Ready MySQL: When MyVector Meets ProxySQL

Sailing Trivia



What Cruisers Do

What society thinks I do

What my mom thinks I do

What my friends think I do

What my colleagues think I do

What I think I do

What I really do

Alkin Tezuysal

Director of Services @AltinityDB

Open Source Database

Evangelist

/askdba

X

@ask_dba

Born to Sail, Forced to Work!

Previously

ChistaDATA, PlanetScale, Percona and Pythian as Senior Technical Manager, SRE, DBA

Earlier in Life

Enterprise DBA , Informix, Oracle, DB2 , SQL Server

- **Recent Recognitions:**
  Most Influential in Database Community 2022 - The Redgate 100
- MySQL Cookbook, 4th Edition 2022 - O'Reilly Media, Inc.
- MySQL Rockstar 2023 - Oracle (MySQL Community)
- Database Design and Modeling with PostgreSQL and MySQL 2024 - <Packt>
- Oracle ACE Pro 2025 - Oracle

What we will cover

Context: MySQL is the system of record; vectors are an add-on.

Goal: keep OLTP predictable while enabling AI retrieval workflows.
Proposal: MyVector (prototype plugin) + ProxySQL as the control plane.
We're actively looking for feedback to drive the next months of work.

MyVector

Vector Plugin for MySQL

Note: this is a design discussion. Names/diagrams represent a WIP implementation and can map to your equivalent components.

Use cases & what “AI queries” mean

Applications are adding semantic retrieval

- Search over text/images/code with meaning, not just keywords
- Recommendations and similarity features
- RAG pipelines (retrieve → generate) become a default pattern

Vector search is the core primitive

- Embeddings map content to high-dimensional vectors
- Similarity search finds nearest neighbors (often ANN)
- The workload is read-heavy, latency-sensitive, bursty

MySQL is already the system of record

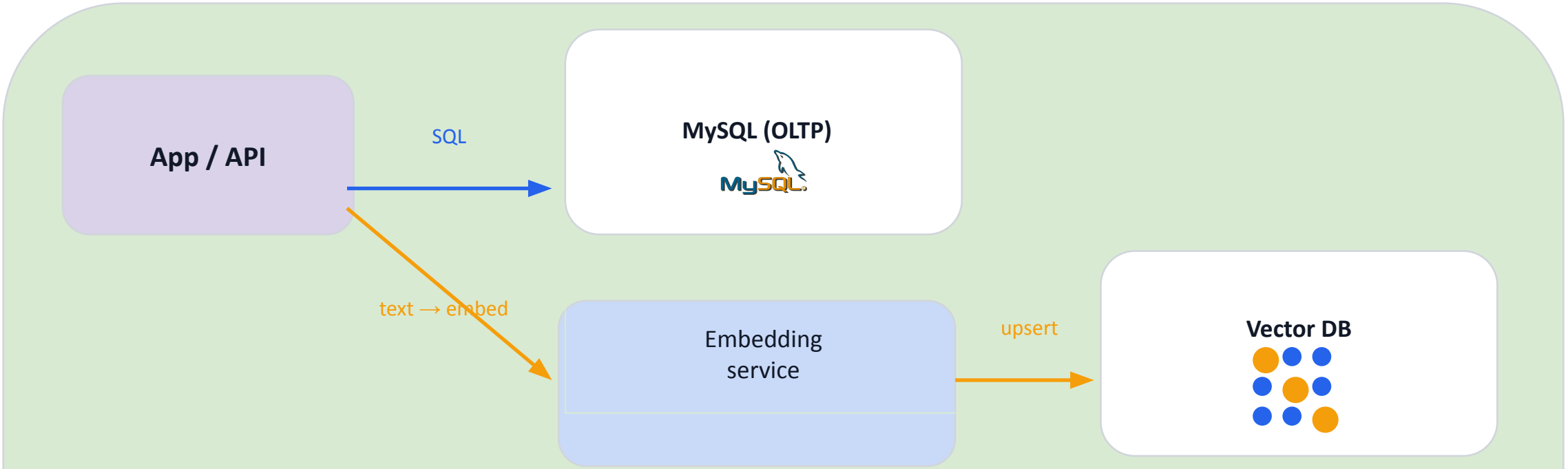
Primary use case: RAG over documentation / knowledge bases.

Secondary: incident/runbook search and internal wikis (SRE-friendly).
Optional: code search (semantic retrieval over code chunks + symbols).

Practical meaning: embed functions/files; retrieve relevant snippets; optionally re-rank with an LLM.

Key constraint: do this without fragmenting data/ops across multiple control planes.

The common “bolt-on vector DB” pattern



Typical pattern: app talks to MySQL for OLTP and a separate vector DB for retrieval.

Pain points we hear from users:

- Two security models, two observability stacks, two SLO conversations.
- Operational drift: schema/data sync, backfills, replay, and consistency gaps.
- Resource surprises: vector bursts overload shared replicas or caches.

We want a simpler, MySQL-native story.

Design goals

Single control plane

Single control plane for SQL + vector workloads (routing, policies, observability).
Strong isolation: protect production OLTP P99 and replication health.
Incremental adoption: start with read-only retrieval on dedicated replicas.
Pluggable storage: vectors can be in-table, side tables/schema, or separate instances.
User-driven roadmap: classification, ingestion, and safety controls should match real workloads.

Protect OLTP stability

Hard isolation so AI bursts do not steal resources from transactions.

Operational clarity

Simple SLOs, clear failure domains, and safe fallbacks.

Extensible

Works with different embedding models and vector index strategies.

Incremental rollout

Start with routing & observability; add vectors and indexes gradually.

Two building blocks

ProxySQL

Orchestration / control plane

MyVector: a MySQL plugin (prototype) for MySQL 8.0 / 8.4 (no native VECTOR type).

Supports vector storage and indexing (focus: retrieval performance and stability).

ProxySQL: orchestration layer for classification, routing, and policy enforcement.

Production-proven in front of MySQL; extends naturally to vector-aware traffic.

MyVector

Vector Plugin for MySQL

Illustrative capabilities:

- Vector data type + distance functions
- Secondary index for ANN or exact search
- Hybrid queries (filters + similarity)
- Operational controls (memory, threads, timeouts)

Key constraint: keep InnoDB stability first.

MyVector Architecture - Vectors

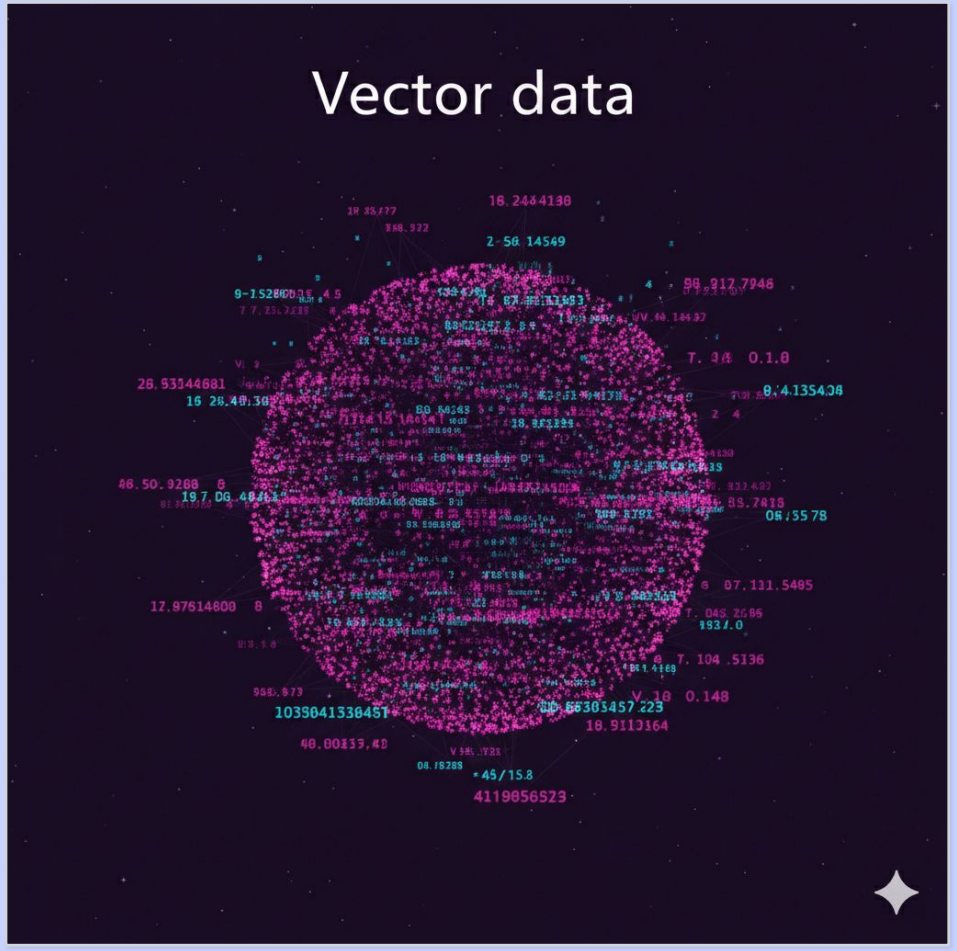
Vectors and Dimensions

Objects Dimensions

Data Points
Mathematical objects representing magnitude and direction (velocity, force)

When dimensions increase data points increase and they turn into embeddings

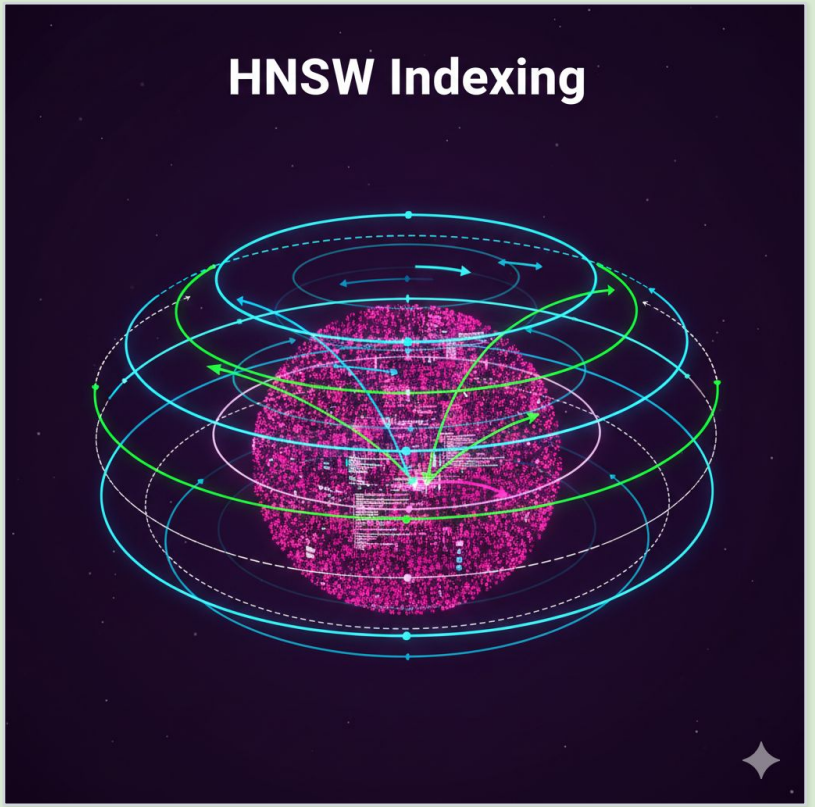
Represented as arrays or ordered lists of numbers



MyVector Architecture - HNSW Indexing

Feature	Brute Force (kNN)	HNSW (ANN)
Accuracy	100% (Exact)	~95-99% (Approximate)
Search Speed	$O(N)$ (Slow)	$O(\log N)$ (Blazing Fast)
Memory Usage	Low	High (Needs to store the graph)
Scalability	Poor	Excellent (Millions of vectors)

Used by: HNSWlib (Hierarchical Navigable Small World)
Source: https://github.com/nmslib/hnswlib
Purpose: High-performance Approximate Nearest Neighbor (ANN) search

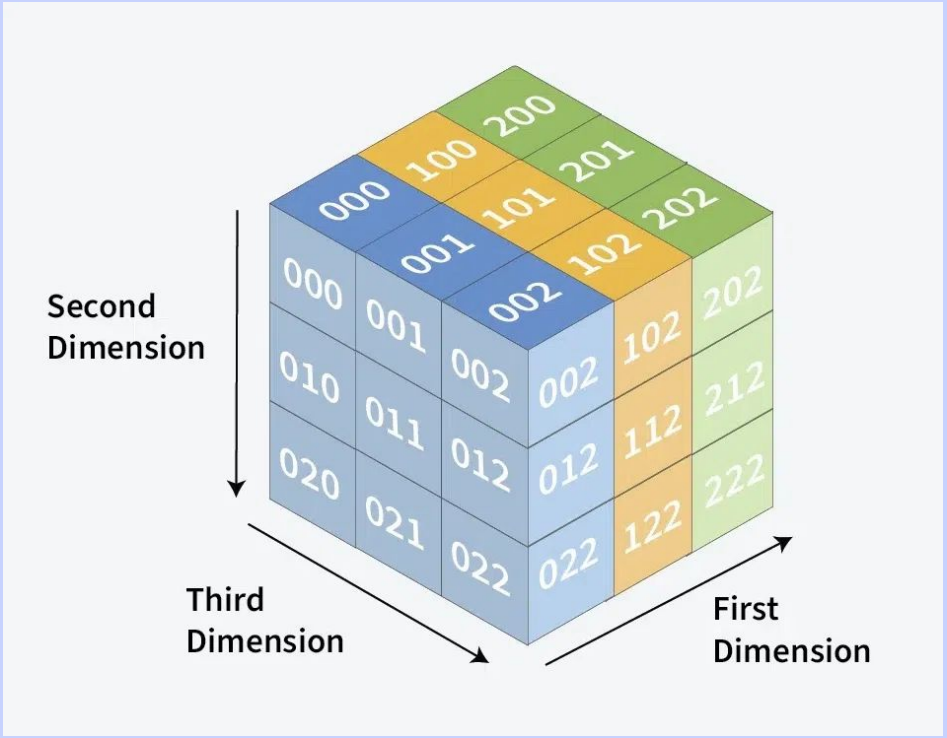


MyVector Architecture - Vectors

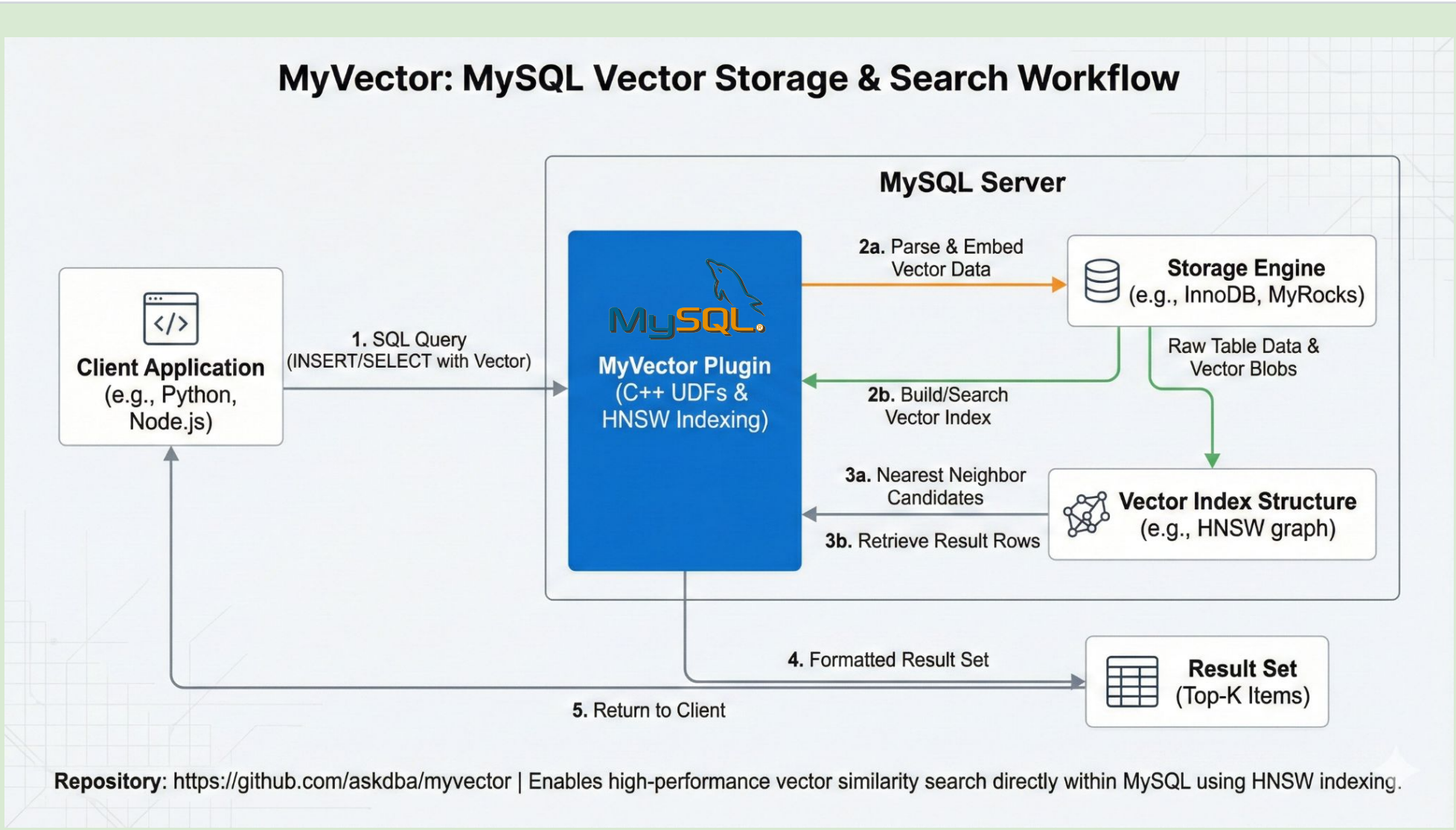
Vector Search
Capture semantic meaning and relationships between words, phrases, or other data.

Pre-Trained
BERT, OpenAI, Word2Vec, CLIP for images

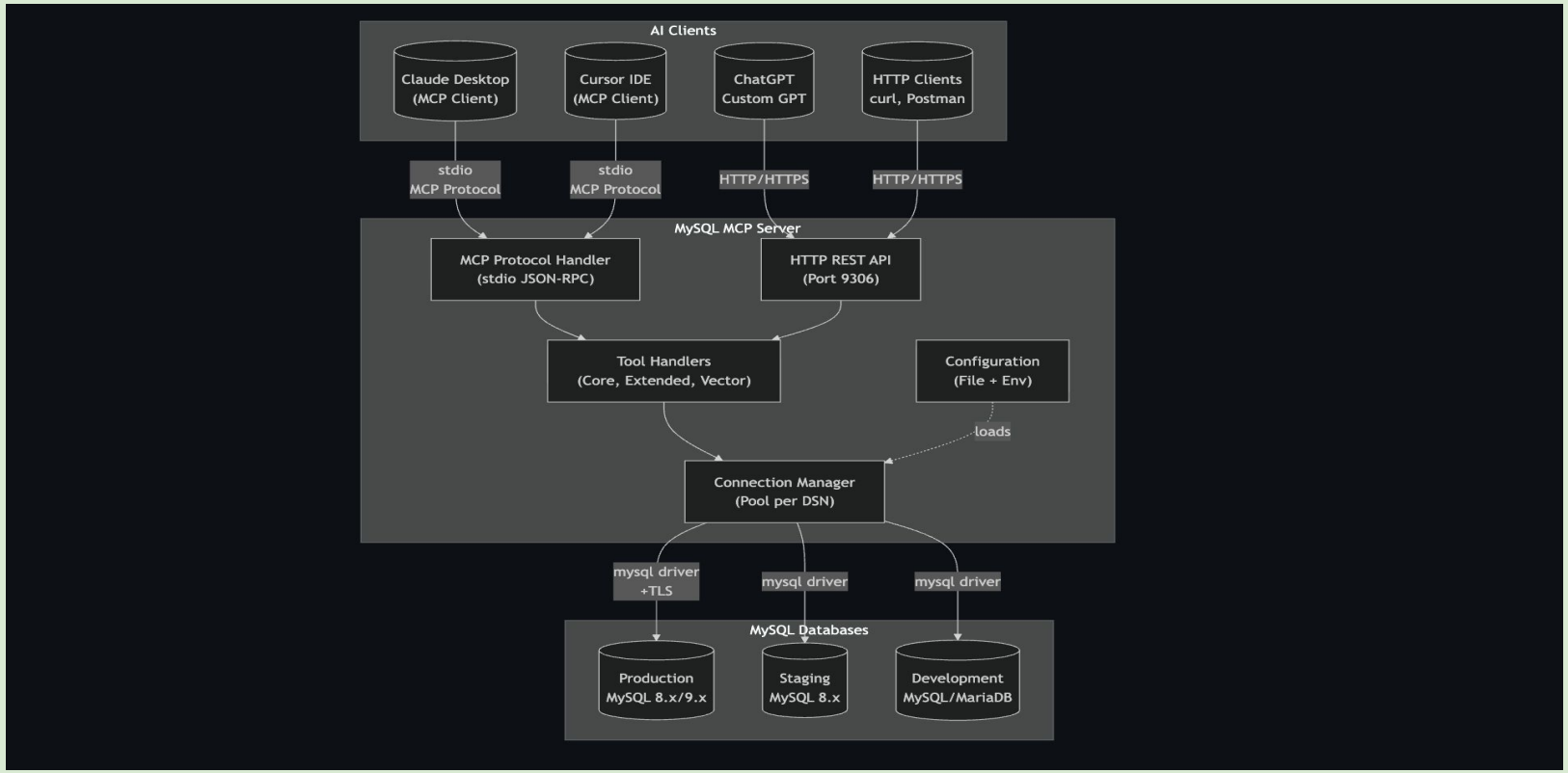
It is a technique that retrieves the most similar items by comparing vector embeddings using distance or similarity metrics in high-dimensional space.



MyVector Architecture

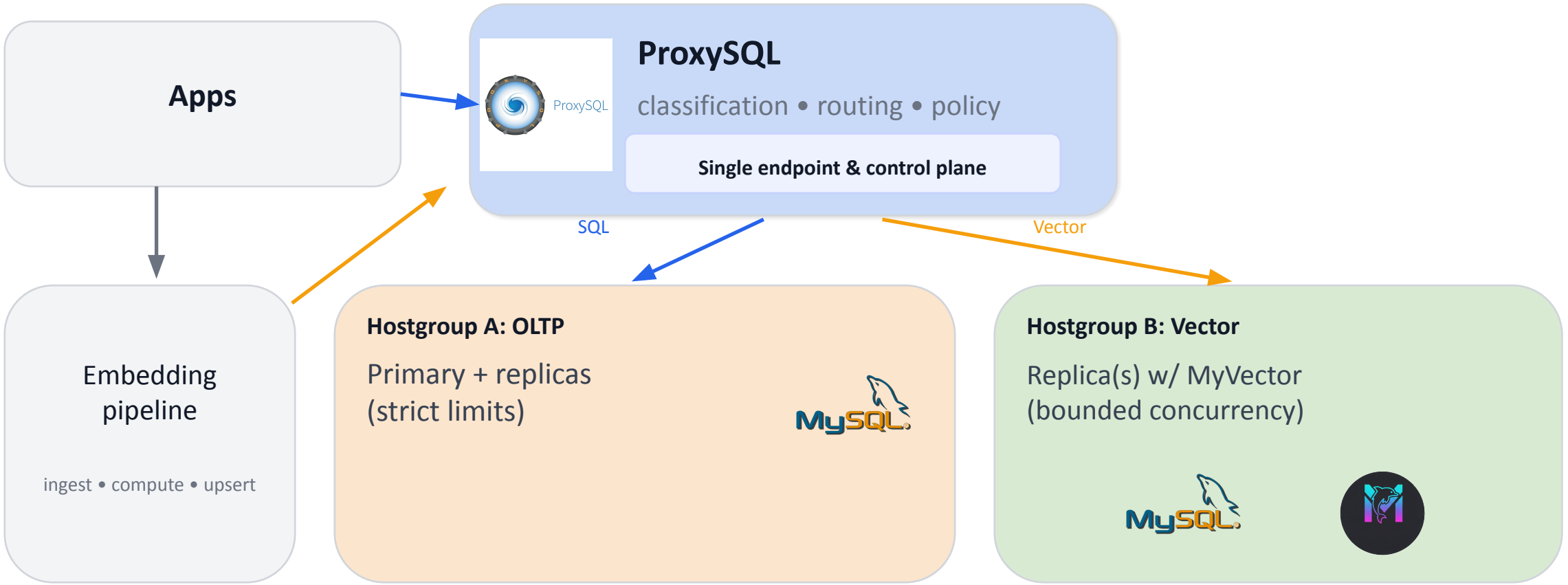


MySQL MCP Server



https://github.com/askdba/mysql-mcp-server

Unified architecture (single control plane)



Apps keep a familiar SQL endpoint (ProxySQL) for OLTP and hybrid queries.
Agents/services can use a dedicated MCP endpoint (still governed by ProxySQL policies).
ProxySQL routes vector-aware reads to the right place:
Dedicated ‘search’ instance or vector replicas, or shared replicas with strict caps.
Embedding pipeline can be orchestrated alongside query traffic (visibility + safety).

Content-aware query classification (ProxySQL)

Rule stack (conceptual):

1. Deny / sanitize unsafe statements (DDL, cross-db reads)
2. Detect vector functions / hints (e.g., VEC_DISTANCE, ANN_SEARCH)
3. Route vector reads → Hostgroup B
4. Route OLTP writes → Hostgroup A primary
5. Route OLTP reads → Hostgroup A replicas
6. Apply per-class limits (timeout, max rows, concurrency)

Example: hybrid query

```
SELECT word, myvector_row_distance() as distance
FROM words50d
WHERE
MYVECTOR_IS_ANN('vectordb.words50d.wordvec',
'wordid', @school_vec, 10);
```

Classification signals:

- function patterns / hints
- read/write intent
- tenant / workload tags

→ route + enforce policy

Resource isolation patterns

Connection & concurrency

- Separate pools per workload class
- Hard caps on vector concurrency
- Queue / shed load under pressure
- Different timeouts & max rows

Compute & IO placement

Isolation is non-negotiable for DBA/SRE audiences.

Patterns we support (choose per environment):

- Dedicated search MySQL instance (strongest protection for OLTP).
- Dedicated vector replicas (separate from ‘regular’ read replicas).
- Shared replicas with hard limits (max concurrency, CPU quotas, timeouts).

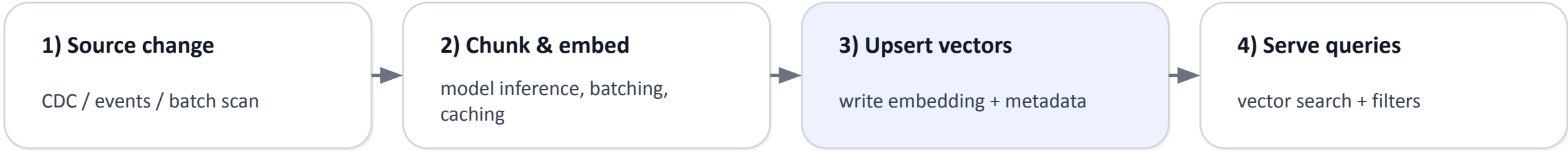
ProxySQL enforces policy consistently and makes overload behavior predictable.

Policy-driven routing

- Tenant-aware limits (gold/silver tiers)
- Feature flags per service
- Allow-list vector functions
- Circuit breakers with safe fallback

Embedding pipeline orchestration

Goal: keep ingestion and refresh safe, observable, and decoupled from OLTP.



Where ProxySQL helps

Embedding ingestion has two viable models:

- Write to primary (strong consistency, simpler reads; higher OLTP risk).
- Write to dedicated replica / replica chain (protect OLTP; design for freshness/HA).

ProxySQL can coordinate the pipeline:

- Queue/backpressure, batching, retry, idempotency, and visibility.

Feedback check: what freshness lag is acceptable (seconds vs minutes vs hours)?

RAG Ingest CLI Tool: Automated Data Pipelines

rag_ingest is a command-line utility for automated RAG index ingestion.

Key Features

MySQL Protocol: Connects via standard MySQL client

Incremental Ingestion: Sync cursors track processed data (watermarks)

Batch Embeddings: Efficient API usage with configurable batch sizes

- Flexible Mapping: JSON-based document transformation and chunking

Main Commands

init — Create RAG schema (tables, FTS5 index, vector tables)

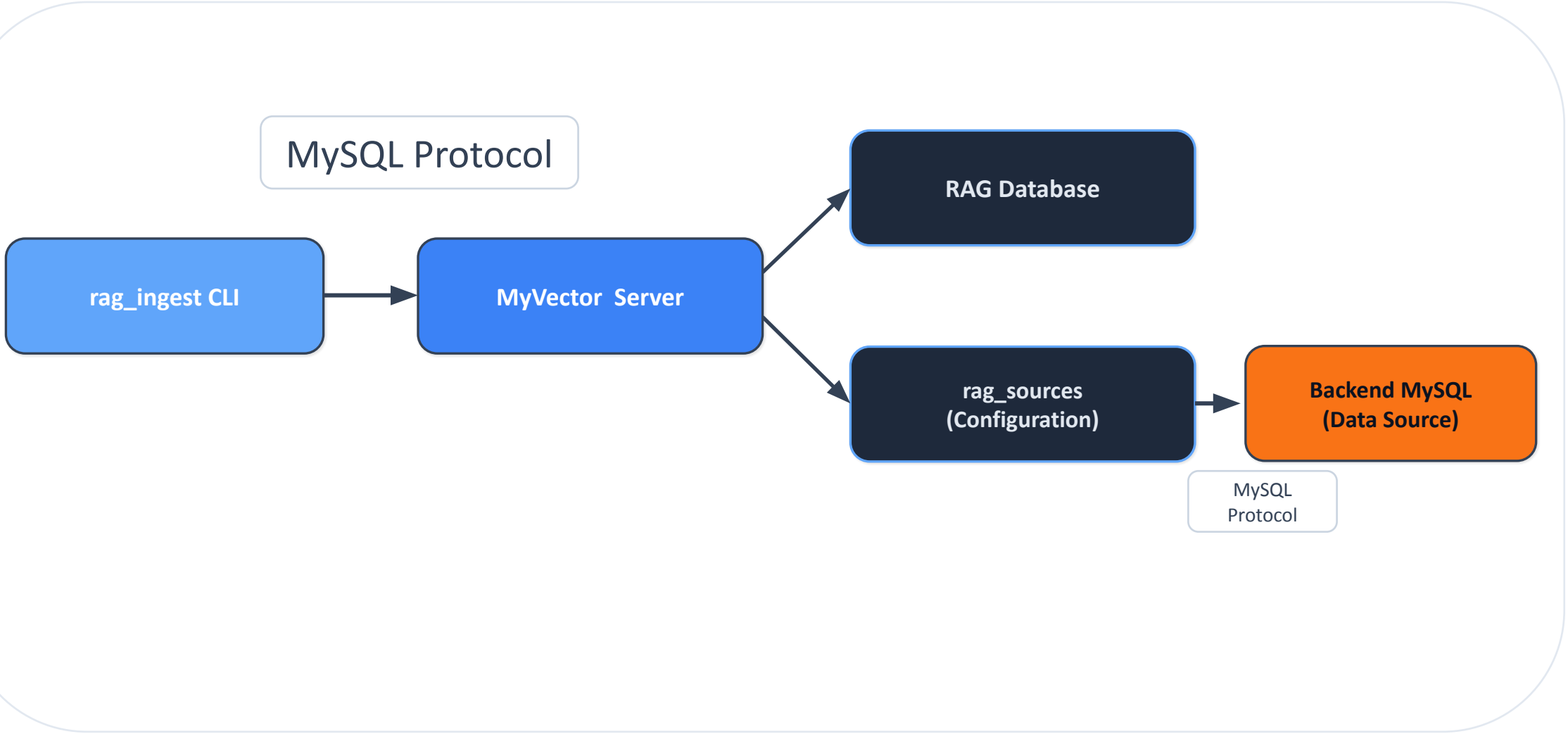
ingest — Process data from configured sources

- query — Vector similarity search

Core Architecture

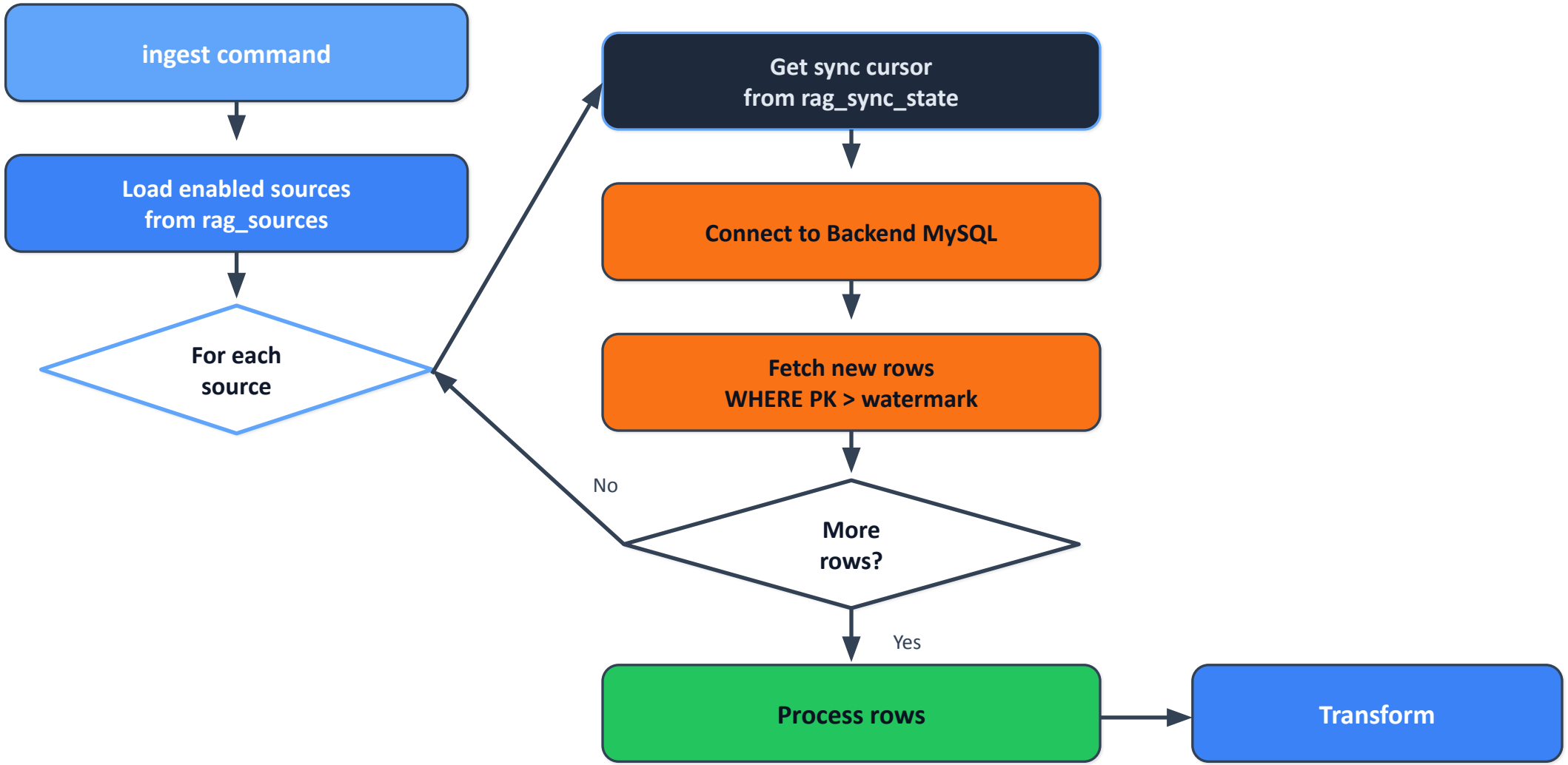
```
1 rag_ingest (CLI) → MyVector Server → RAG
Database
2                                      →
Backend MySQL (source)
```

RAG Ingest: Architecture Overview



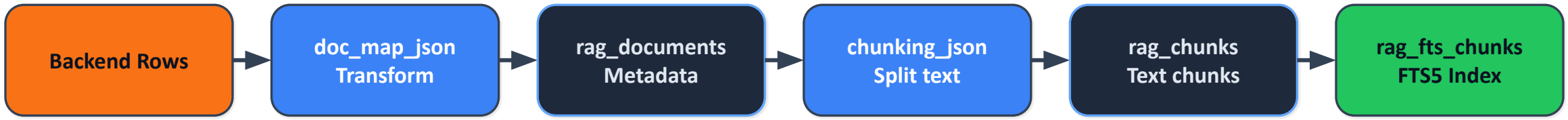
CLI connects via MySQL protocol to MyVector Server, which manages the RAG database and connects to backend MySQL sources.

RAG Ingest: Step 1 – Fetch Data



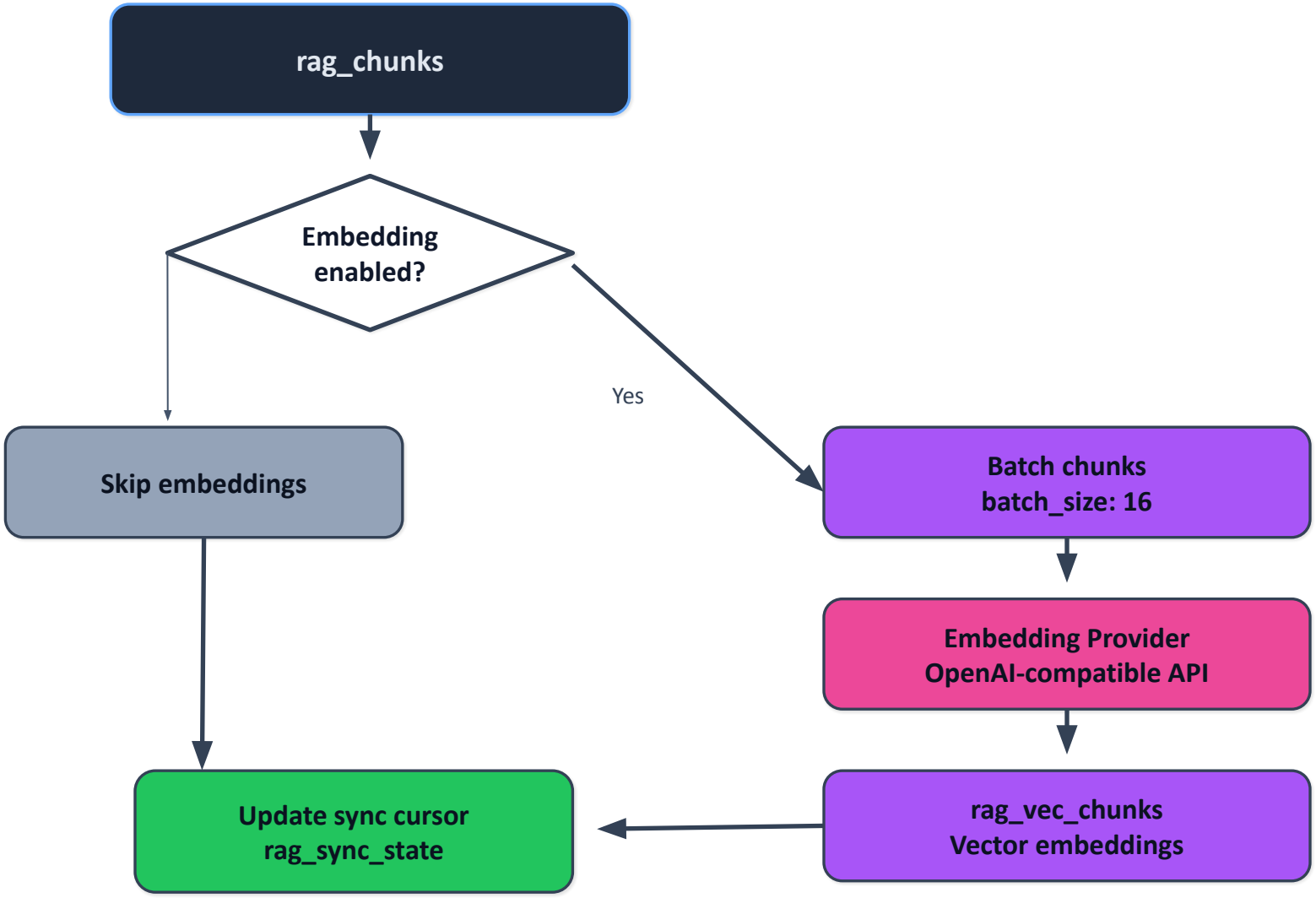
Fetch stage: Load sources → Get watermark → Connect to backend → Fetch incremental data

RAG Ingest: Step 2 – Transform & Chunk



Processing: Transform rows → Document metadata → Chunk text → FTS index

RAG Ingest: Step 3 – Embeddings (Optional)



Optional: If enabled, batch chunks and generate embeddings via OpenAI-compatible API

RAG Ingest: Data Source Configuration

Configure data sources in the rag_sources table:

Configuration Components

doc_map_json: Transform backend rows → RAG documents

chunking_json: Text splitting (chars, chunk_size, overlap)

- embedding_json: Vector generation (provider, model, batch_size)

Supported Providers

stub — Pseudo-embeddings for testing

- openai — OpenAI-compatible APIs (OpenAI, Azure, local models)

Example Flow

```
INSERT INTO rag_sources (name, enabled,
  backend_type, backend_host, backend_port,
  table_name, pk_column, where_sql,
  doc_map_json, chunking_json, embedding_json
) VALUES ('blog_posts', 1, 'mysql', '127.0.0.1',
3306,
  'posts', 'Id', 'Published = 1',
  {...}, {...}, {...});
```

Then run: `rag_ingest ingest`

RAG Ingest: Incremental & Batch Processing

Incremental Ingestion (Watermarks)

Sync cursors stored in rag_sync_state track last processed PK

Only processes new rows (PK > watermark) on each run

- Existing documents (matching doc_id) are automatically skipped

Production Features

Configurable log levels (error, warn, info, debug, trace)

Authentication via ProxySQL mysql_users

- Network isolation support (bind to 127.0.0.1)

Batch Embedding Generation

```
1 {"batch_size": 16}
```

- 100 chunks ≈ 7 API calls (instead of 100)
- Reduces latency and network overhead
- Configurable timeout and retry logic

Operational safety: make failure modes boring

Traffic & reliability

- SLOs per class: OLTP p99, vector p95, ingestion lag
- Unified observability at the proxy: query tags, hostgroups, retries
- Circuit breakers: shed vector load before OLTP suffers
- Progressive rollout: shadow reads → small tenants → default

Database safety

Make failure modes boring (and easy to debug).

Examples of guardrails:

If vector hostgroup is degraded: fail fast, fallback, or serve cached results.

Tripwires: replication lag thresholds, memory pressure, CPU saturation.

Sane defaults: deny unknown vector ops; explicit enablement per tenant/service.

Goal: when retrieval breaks, OLTP stays healthy and the blast radius is contained.

Example tripwire policy

If OLTP p99 > target for 2 minutes → cut vector concurrency by 50% and reject non-critical vector queries.

If ingestion lag > threshold → pause refresh for low-priority tenants.

Security & governance (often overlooked)

Security and governance matter more once LLM agents are involved.

Controls to consider:

- Per-tenant/service quotas for vector queries and embedding writes.
- Allow-list which schemas/tables can be retrieved through the MCP endpoint.
- Audit trails: who searched what; sampling for sensitive data exposure.

Feedback check: which governance controls are must-have for you?

Practical rule: treat “vector search” as a new API surface, not just a new index.

A pragmatic migration path

Phase 0

Introduce ProxySQL as the traffic control point

Centralized routing, pooling, and observability. No vectors yet.

Phase 1

Store embeddings alongside data (no ANN)

Add vector column + simple distance UDFs for low-volume use cases.

Phase 2

Add MyVector index + isolate vector hostgroup

Replica(s) serve similarity queries with tight policy limits.

Phase 3

Operationalize: freshness, backfills, tiering

Embedding refresh, re-indexing, and tenant-aware QoS become routine.

Vector storage options in MySQL

Option A: In-table VECTOR column (Simplest Query)

- Pros: Single table, easy joins, fewer moving parts.
- Cons: Requires schema change; might increase row size/page churn.

Option B: Side table keyed by PK (Recommended Default)

- Pros: Minimal change to canonical tables; separate tuning.
- Cons: Requires join; manage lifecycle (delete/update).

Option C: Separate schema or MySQL instance (Strong Isolation)

- Pros: Isolates storage/load; independent maintenance.
- Cons: Needs sync/backfill strategy; more operational overhead.

Endpoint strategy: SQL + MCP without chaos

SQL endpoint (classic): apps send normal SQL and hybrid queries via ProxySQL.
MCP endpoint (agents): tool-based access, still governed by ProxySQL policies.
Why separate endpoints?

- Different trust model: agents may generate unexpected queries.
- Different rate/limits: MCP can burst; needs stricter caps.
- Different auditing: agent activity must be traceable.

Same control plane: both endpoints map to hostgroups + policy sets.

Observability we should add (vector-aware)

Goal: make vector workloads debuggable with the same rigor as OLTP.

ProxySQL metrics/extensions (ideas):

- Vector query class (top-k, hybrid filter, re-rank, batch).
- k distribution, filter selectivity, index hit/miss or fallback-to-exact.
- Latency by class + queue time (backpressure visibility).
- Concurrency saturation, breaker trips, shed-load counts.

MySQL/MyVector metrics (ideas):

- Index build/maintenance time, memory footprint, compaction stats.
- Vector freshness lag, ingest backlog / queue depth.
- Replica lag correlated to vector bursts.

Classification: detection vs explicit tagging

Option 1: automatic detection (functions/operators) — lowest app change.

Risk: false positives/negatives; start conservative with allow-lists.

Option 2: explicit tagging (SQL comments / hints) — precise intent.

Risk: requires app discipline; tags must be validated/authorized.

Option 3: session attributes (service/tenant identity) — policy-first.

Risk: identity propagation must be reliable; still needs parsing.

Recommended compromise:

Start with detection + strict allow-list; add optional explicit tags for power users.

Policy examples (what ProxySQL can enforce)

Routing:

- Vector reads → vector hostgroup; OLTP writes → primary hostgroup.
- Hybrid queries → vector hostgroup with strict timeouts.

Protection:

- Max concurrent vector queries per service; per-user throttles.
- Hard timeouts + query kill; cap k and result size.
- Circuit breaker: if replica lag > threshold, shed vector load.

Governance:

- Allow-list vector tables/schemas; block dangerous joins.
- Audit log: who searched what via MCP.

Failure-mode story: vector overload without OLTP impact

Scenario: an agent triggers a burst of retrieval queries (k=200) at peak traffic.

Control plane response:

- ProxySQL classifies vector workload → routes only to vector hostgroup.
- Concurrency caps queue requests; timeouts prevent runaway latency.
- Tripwire sees replica lag rising → breaker sheds/limits vector traffic.
- OLTP hostgroup unaffected; writes and critical reads stay within SLO.

Outcome: retrieval degrades gracefully; production stays healthy.

Embedding lifecycle: backfill vs incremental updates

Two phases most teams underestimate:

Initial backfill:

- Compute embeddings for existing corpus; heavy IO/CPU; needs batching and throttling.
- Prefer off-peak or dedicated pipeline; track progress; retry idempotently.

Incremental updates:

- Trigger on writes (CDC/binlog) or app events.
- Re-embed on content change; delete embeddings on row deletion.

Operational requirement: idempotency + deduplication + auditability.

Where should embedding writes land?

Model A: write embeddings to primary (strong consistency).

- Best when: small vectors, low update rate, strict freshness needs.
- Risk: adds write load; can hurt OLTP and replication.

Model B: write to dedicated replica/chain (strong isolation).

- Best when: large corpora, frequent backfills, strict OLTP protection.
- Risk: eventual consistency; needs HA plan for embedding store.

Design goal: allow both, controlled by policy and environment.

What feedback we need (design partner input)

Workload characteristics:

- Corpus size (rows/vectors), vector dimensionality, update frequency.
- Query patterns: top-k, hybrid filters, re-ranking, batch sizes.

SLO / constraints:

- Target P95/P99, acceptable freshness lag, acceptable recall trade-offs.
- Hard constraints: replication lag ceilings, maintenance windows, cost budgets.

Governance:

- Tenant isolation needs, auditing requirements, data sensitivity.

Even anonymized numbers will directly shape sensible defaults.

Example schema: side table for embeddings

Canonical table (unchanged): docs(id, title, body, updated_at, …)
Embedding table (new): doc_embeddings(
    doc_id PK/FK → docs.id
    model_id (for migrations)
    embedding (plugin-managed vector storage)
    embedded_at timestamp
    metadata (tenant, tags, source)
)
Indexing: MyVector index on embedding + optional partitioning by tenant/model.

Example queries: RAG retrieval (conceptual)

1) Embed the question (service or client library).
2) Retrieve candidates (IDs):

```
SELECT doc_id FROM doc_embeddings
WHERE tenant_id = ?
ORDER BY vec_distance(embedding, ?) ASC
LIMIT k;
```

3) Fetch docs by doc_id; compose prompt with citations.

Operational checklist (early adopters)

- Capacity planning: separate vector replicas/instance; reserve CPU/memory.
- Limits: cap k, enforce timeouts, concurrency limits per service.
- Backfill plan: throttled, resumable, idempotent.
- Monitoring: lag tripwires, queue depth, vector latency class.
- Security: allow-lists, separate MCP credentials, auditable logs.

Key takeaways

MyVector adds MySQL-native vector capability (prototype, evolving quickly).

ProxySQL provides the control plane: classification, routing, isolation, and governance.
The core promise: AI retrieval without sacrificing MySQL stability.
If you have workloads (RAG, runbooks, code search), we want to learn from you.

Q&A

Contact: <alkin.tezuysal@gmail.com> | Repo/Docs: <https://github.com/askdba/myvector>
Contact: <rene.cannao@gmail.com> | Repo/Docs: <https://github.com/ProxySQL/proxysql-vec>

References:

- **GitHub - sysown/proxysql: High-performance MySQL proxy with a GPL license.**
- GitHub - askdba/mysql-mcp-server: MySQL Server implementation for Model Context Protocol (MCP)
- written in Go.
- GitHub - askdba/myvector: Vector Storage & Search Plugin for MySQL
- https://dev.mysql.com/doc/refman/9.6/en/vector.html?st_source=ai_mode#:~:text=MySQL%20Enterprise%20Edition%20%C2%B7%20MySQL%20Workbench%20%C2%B7%20MySQL%20on%20OCI%20Marketplace%20%C2%B7%20Telemetry&text=MySQL%20Community%20Server%20%C2%B7%20MySQL%20NDB%20Cluster%20%C2%B7%20MySQL%20Shell%20%C2%B7%20MySQL%20Router