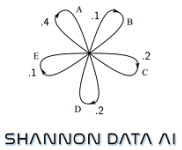
ShannonBase





SHANNONBASE

The next-Gen Database for AI/ML

The next-generation database for the AI era — unified HTAP, in-database machine learning, vector search and generative AI on the MySQL ecosystem

Business / Technical Whitepaper

Copyright © 2026, Shannon Data AI

Contents

Audience 5
Executive Summary 6
Challenges in Building a Modern Data Platform 7
Our Solution: ShannonBase 8
Architecture Overview 8
Dual-Engine Design Principles 9
HTAP and Intelligent Query Routing 11
Query Optimization and the Cost Model 12
Real-Time Change Propagation 12
The Rapid Engine — In-Memory Columnar Storage 14
Three-Level Storage Architecture 15
Storage Index 16
MVCC and Transactional Consistency 16
Vectorized Execution and SIMD Acceleration 17
Parallel Query Execution (Morsel) 17
Checkpoints and Recovery 17
Resource Governance and Observability 18
Parallel Load 18
Autopilot — The Self-Managing Memory Engine 18
Table Importance Scoring 20
Time-Based Importance Decay 20
Four-Phase Auto-Loading Algorithm 21
Memory Threshold Management 21
System Quiet Detection 21

Copyright © 2026, ShannonBase Project Contributors / Public

User-Load Protection ... 22
Background Coordinator Thread ... 22
Metadata Bootstrap ... 22
Autopilot Capabilities at a Glance ... 23
AutoML — In-Database Machine Learning ... 24
The Challenge of Machine Learning on Database Data ... 24
How AutoML Works ... 24
Usage Example ... 25
Training and Inference Performance ... 26
Vector Search and Generative AI ... 26
Native VECTOR Data Type ... 26
In-Database Embedding Generation ... 27
In-Database LLM Inference ... 28
RAG Architecture ... 28
JavaScript Multilingual Engine (MLE) ... 30
Native-Agent Inside the Database ... 31
Design Motivation: Closing the Gap Between Data and Intelligence ... 32
Architecture ... 33
Intent Routing and Execution Paths ... 34
Conversation Memory and Multi-Turn Context ... 35
Schema-Aware SQL Generation ... 37
Transactional Write Safety ... 37
Plugin Extension Model ... 38
Direct Access to ML Primitives ... 40
Integration with the ShannonBase Architecture ... 41

Copyright © 2026, ShannonBase Project Contributors / Public

Typical Use Cases 42
Security and Data Governance 43
Native-Agent Capabilities at a Glance 43
ShannonBase Deployment 44
Deployment Models 45
System Variables and Configurability 45
Security, Observability and Manageability 46
Target Workloads and Use Cases 47
Use Cases 47
Conclusion 48
Resources 48
Contact Us 49

Audience

This document is part of the ShannonBase technical whitepaper series. It describes the basic technical principles and concepts of ShannonBase, an open-source HTAP database management system built on MySQL 8.4.11 LTS.

It is written for engineers, architects and technical decision makers, and explains the system architecture, the Rapid in-memory column engine, change propagation and recovery, AutoML, vector search and generative AI, JavaScript MLE, Native-Agent, and the deployment models and system variables behind them.

Copyright © 2026, ShannonBase Project Contributors / Public

Executive Summary

ShannonBase is an open-source, HTAP (Hybrid Transactional/Analytical Processing) database management system built on the MySQL ecosystem. The current release is v0.1.0, built on the MySQL 8.4.11 LTS kernel. Within a single instance it provides OLTP transaction processing, real-time OLAP analytics, in-database machine learning (AutoML), a vector data type with vector similarity search, generative AI inference, and a Native-Agent built on the JavaScript agent runtime. All of these are reached through standard MySQL interfaces. This release targets single-node HTAP plus horizontal read-replica scaling; the clustered form that shares Rapid memory across nodes is planned for the enterprise edition.

By embedding Rapid — an in-memory columnar storage engine developed in-house — alongside InnoDB, ShannonBase achieves true HTAP through transparent query routing: analytical workloads are offloaded automatically to the column engine while transactional operations continue on the row store. The two engines are kept in sync through DML Notification (the default) or COMMITTED_BINLOG (decoding of committed binlog); parsing the redo log for synchronization is also supported, for cloud-native architectures. Analytical queries use the same ReadView as InnoDB together with the change watermark required by the query snapshot, and never read uncommitted or partially applied transactions. All Rapid data can be rebuilt completely from InnoDB plus binlog, so the column engine provides acceleration only — never durability or commit.

ShannonBase positions itself as the MySQL of the AI era: it delivers the familiar MySQL interfaces, protocol and ecosystem while providing the analytical depth, machine-learning breadth and generative AI capability that a modern enterprise data platform requires.

Copyright © 2026, ShannonBase Project Contributors / Public

Challenges in Building a Modern Data Platform

Organizations building modern data platforms face the challenges inherent in traditional architectures: a set of problems that must be solved without significantly increasing complexity, cost or operational burden:

- **Data complexity: explosive data growth, a diversity of formats (structured, semi-structured, unstructured) and heterogeneous data sources create an ETL (Extract/Transform/Load) integration burden that cannot scale.**
- **Fragmented services: traditional architectures separate workloads into independent systems — OLTP (Online Transactional Processing) databases, OLAP (Online Analytical Processing) data warehouses, ML (Machine Learning) platforms, vector databases, LLM (Large Language Model) services and more — each requiring its own resource allocation, security policy and operational lifecycle.**
- **Real-time data processing: traditional ETL introduces latency between the moment a transaction occurs and the moment its data can be analysed. Modern applications demand real-time analytics over live transactional data.**
- **High total cost of ownership (TCO): running many specialized services multiplies infrastructure cost, engineering effort and operational complexity. When LLMs depend on external services, generative AI workloads add further GPU cost.**
- **Data security and governance risk: every data movement across a system boundary widens the attack surface and introduces compliance risk — standards such as GDPR strictly limit data leaving its domain.**

Copyright © 2026, ShannonBase Project Contributors / Public

Our Solution: ShannonBase

ShannonBase is a fully open-source, production-ready database management system that lets organizations run OLTP transactions, real-time OLAP analytics, in-database machine learning, vector similarity search and generative AI inference — all through a single SQL statement, with no ETL and no data movement.

Architecture Overview

ShannonBase uses a layered, modular architecture that extends the MySQL kernel with components dedicated to columnar analytics, machine learning and AI inference:

Layer	Component
SQL interface layer	MySQL-compatible parser, DDL/DML processor, standard ODBC/JDBC connectors
Query processing layer	MySQL parser and optimizer extended with an HTAP cost model and secondary-engine routing hooks
HTAP routing layer	PrepareSecondaryEngine, ML offload routing, RapidOptimize plan rewriting, cost comparison and safe fallback
Execution layer	InnoDB row-store executor (OLTP path); Rapid batch (vectorized) iterators: scan/filter/join/aggregate/sort/window, with morsel parallelism
Storage layer	InnoDB engine; Rapid IMCS: ART indexes, Storage Index, dictionary/Bit Pack/compression, column-level MVCC versions
Change propagation	DML Notification or COMMITTED_BINLOG (committed binlog decoding) → Rapid IMCU writes
Autopilot layer	SelfLoadManager: importance scoring, decay, priority-queue load/evict, quiet detection, schema bootstrap
ML and AI layer	AutoML (LightGBM) + ONNX Runtime (embedding/generation/PII) + native VECTOR type and DISTANCE functions

Copyright © 2026, ShannonBase Project Contributors / Public

MLE layer	JerryScript JavaScript engine (can execute SQL and emit result sets)
Agent engine	sys.shannon_chat: 34 tools, L1–L4 plugin dispatch, transactional write safety and approvals

Dual-Engine Design Principles

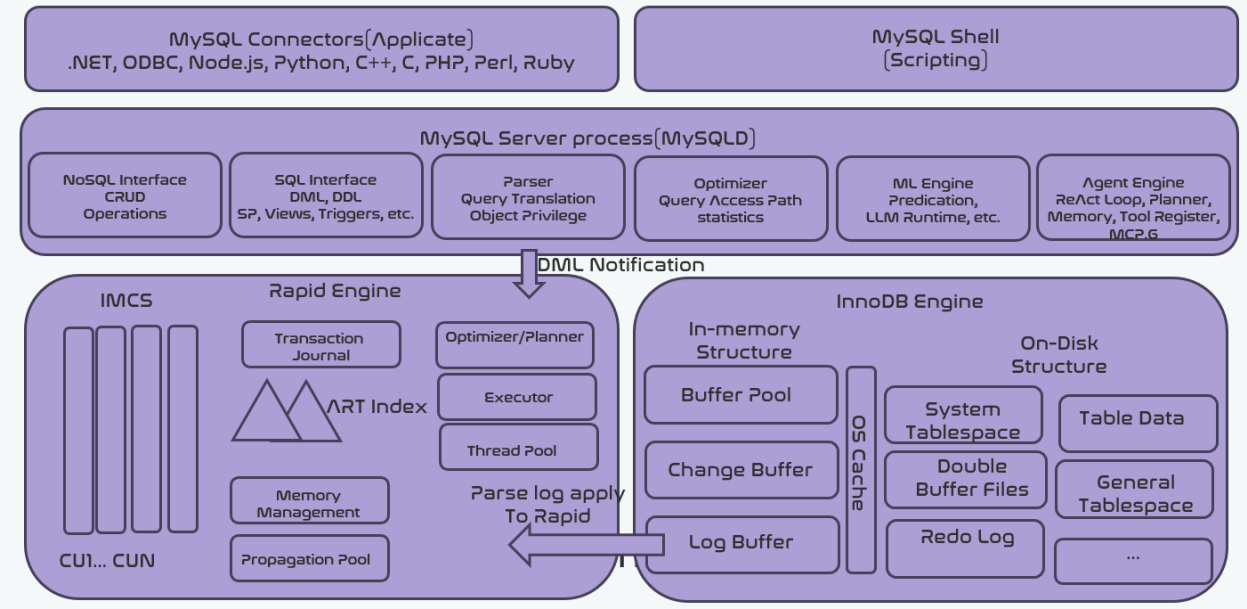
- Transparency: applications use standard MySQL SQL with no code changes; engine selection is invisible to users.
- Consistency: MVCC built on the InnoDB ReadView and the propagation watermark guarantees that analytical results always match the statement's transaction snapshot.
- Intelligence: a cost-model-based routing classifier selects the best engine for every query and adapts dynamically to data readiness and query complexity.
- Efficiency: vectorized execution, SIMD, Storage Index skipping, bit-packed columns and dictionary masks maximize analytical throughput per unit of memory.
- Safety: all Rapid data can be rebuilt from InnoDB and binlog; durability and commit always remain with InnoDB.
- **Recoverability: Rapid uses immutable columnar checkpoints (manifest v3, two generations retained) plus committed-binlog replay for fast restarts; if a checkpoint is missing or fails validation, Rapid falls back to reloading from InnoDB, and the table serves no Rapid reads until recovery completes.**

The system is built on the MySQL code base and is 100% compatible with the MySQL protocol, SQL syntax, ACID semantics and the ecosystem of connectors, ORMs and management tools. Existing MySQL applications can adopt ShannonBase analytical and AI capabilities incrementally without any application-layer change.

Module	Function	Key value
HTAP engine	Intelligent dual-engine query routing between InnoDB and Rapid	Real-time analytics without ETL
Rapid engine	In-memory column store with ReadView MVCC,	About an order of magnitude

Copyright © 2026, ShannonBase Project Contributors / Public

	vectorization, ART indexes, Storage Index and bit-packed columns	speedup on TPC-H queries
Change propagation	DML Notification or COMMITTED_BINLOG decoding for real-time InnoDB→Rapid sync	Millisecond data freshness (waiting to the query watermark)
AutoML	In-database model training, inference and explainability through SQL	No data export, no Python
Vector search	Native VECTOR type, similarity functions and hybrid SQL queries	In-database RAG and semantic search
JavaScript MLE	In-database JS stored procedures and functions on JerryScript	In-database ETL, feature engineering and AI post-processing
GenAI / LLM	In-process embedding and LLM inference (local ONNX Runtime, Ollama or cloud API)	Data sovereignty by choice: local inference never leaves the instance



ShannonBase Architecture Diagram

HTAP and Intelligent Query Routing

ShannonBase delivers HTAP through a dual-engine architecture: InnoDB is the primary row store for transactional workloads and the Rapid engine is the secondary in-memory column engine for analytical ones. The MySQL 8.x secondary engine interface is implemented as four hooks:

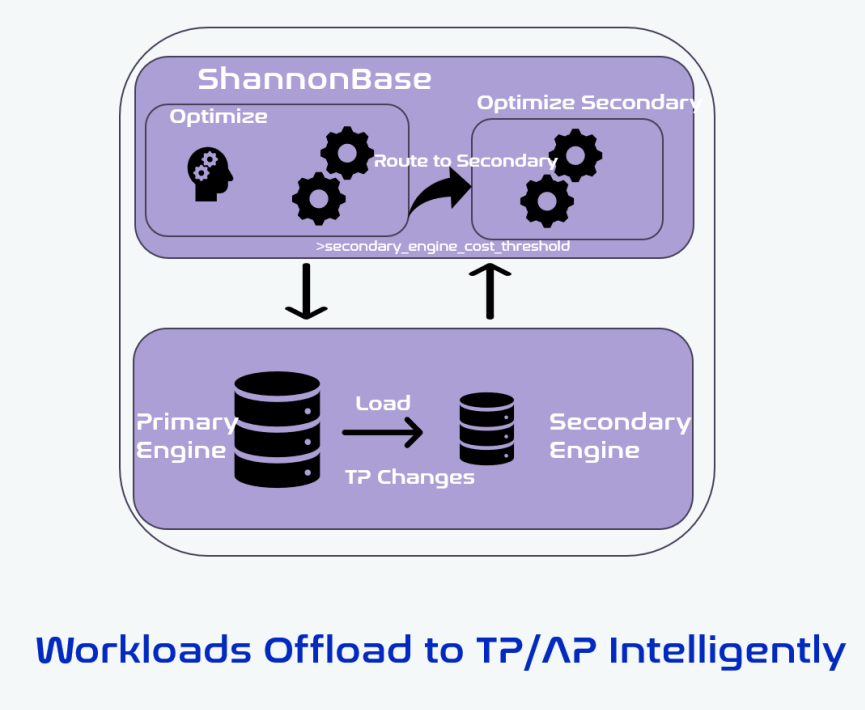
(1) PrepareSecondaryEngine (hard preconditions and resource admission),

(2) SecondaryEnginePrePrepareHook (offload decision based on a cost threshold or an ML classifier),

(3) optimize_secondary_engine (RapidOptimize plan rewriting) and

(4) compare_secondary_engine_cost (Rapid cost estimation).

Only statements at READ COMMITTED or REPEATABLE READ isolation are eligible for offload; FORCED mode skips the cost comparison but still runs every hard check.



Routing decisions are completely transparent to applications. Standard SQL submitted through a MySQL connector reaches ShannonBase, which selects the optimal execution path automatically — no hints, no configuration changes. If Rapid has not yet caught up to the propagation watermark required by the query snapshot, if the target table is quarantined, or if the Rapid plan cannot be converted into a complete query path, the statement falls back to the original MySQL/InnoDB plan,

Copyright © 2026, ShannonBase Project Contributors / Public

which guarantees correctness.

Workload type	Execution engine
Point queries, INSERT/UPDATE/DELETE	InnoDB (row store, MVCC, B+ tree indexes)
Full scans, aggregations, complex joins	Rapid engine (in-memory columnar, vectorized)
Mixed HTAP queries	The optimizer routes each query automatically
Rapid behind the required watermark, or the table is quarantined	Automatic fallback to InnoDB (safety guarantee)

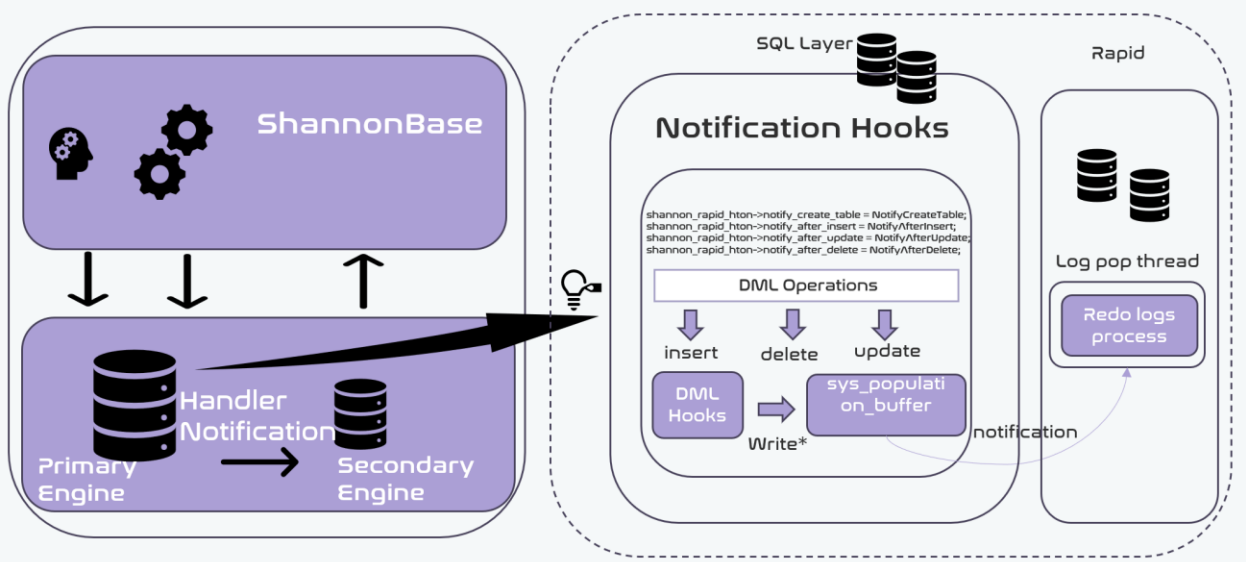
Query Optimization and the Cost Model

The Rapid optimizer runs on top of the AccessPath produced by MySQL: it first translates Item conditions into Rapid predicates, then applies rules in a fixed order — predicate pushdown, Storage Index pruning, projection pruning, LIMIT/OFFSET pushdown, aggregate pushdown (only when join type and multiplicity prove equivalence), cost re-estimation and join reordering. The cost model uses IMCS statistics (row counts, NDV, histograms, quantiles, per-IMCU zone maps) and prices two-phase reads, spilling and parallelism separately. If the Rapid plan cannot be converted into executable iterators, the optimizer keeps the original MySQL plan.

Real-Time Change Propagation

ShannonBase offers two change propagation modes, selected by rapid_change_propagation_mode (which can only be switched while no table is loaded):

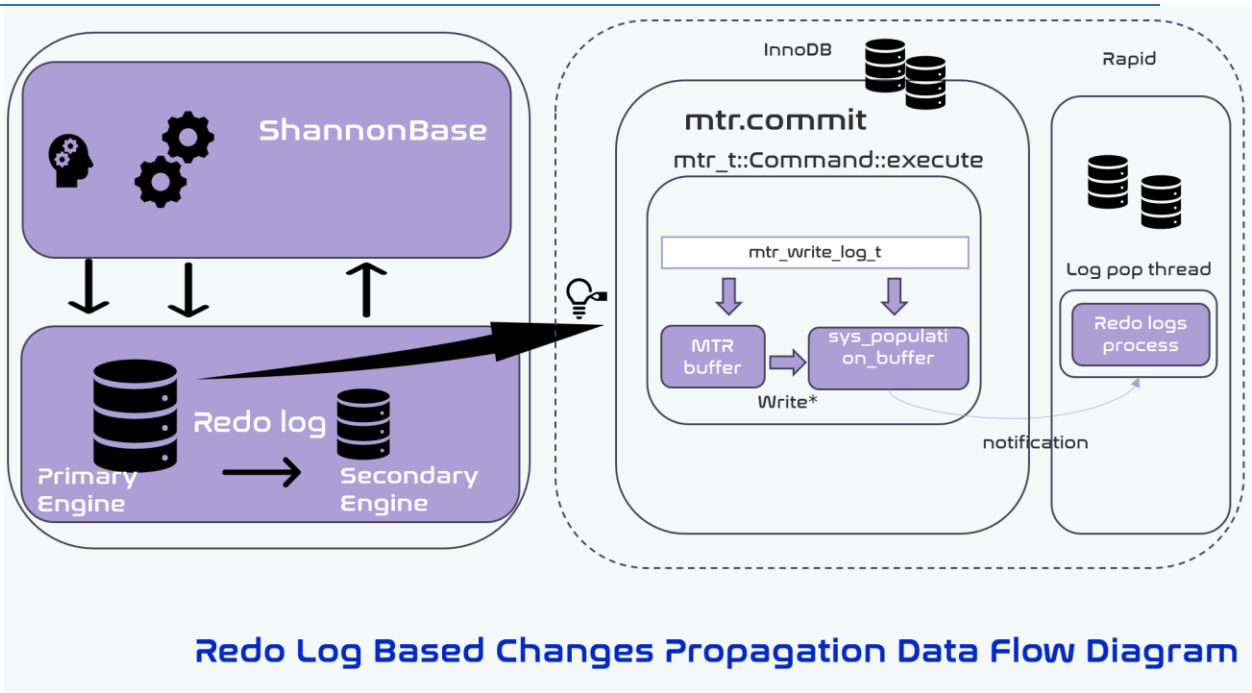
Copyright © 2026, ShannonBase Project Contributors / Public



DML Notification Based Changes Propagation Data Flow Diagram

- CAPTURE (default, DML Notification): the InnoDB DML notification hooks (NotifyAfterInsert/Update/Delete) pack the new row image, the old row image and off-page BLOB data into change records before commit and write them into a 16-shard ring buffer; propagation worker threads apply these records asynchronously. Records carry the source InnoDB transaction ID and enter Rapid MVCC as ACTIVE versions; the outcome of the source transaction's commit or rollback is published by SourceTxnTracker, which decides whether the version is finally committed or undone. Under normal write load, propagation latency is in the single-digit millisecond range.

Copyright © 2026, ShannonBase Project Contributors / Public



Redo Log Based Changes Propagation Data Flow Diagram

- COMMITTED_BINLOG: a background binlog reader decodes committed source transactions and builds row records in the same format as CAPTURE. This mode suits read replicas or deployments that want to be fully decoupled from the source transaction path. It requires a usable binlog configuration (for example binlog_order_commits=ON); when the configuration is not satisfied, the affected tables report an explicit error.

Neither mode changes the outcome of the source transaction: a Rapid-side write failure only stops that table from being read by the secondary engine — analytical queries on the table are simply no longer offloaded to Rapid — and never fails a committed source transaction. Analytical queries acquire the InnoDB ReadView in a fixed order at start-up, capture the table's current propagation watermark and wait only up to that watermark, so they never read uncommitted or partially applied transactions.

The Rapid Engine — In-Memory Columnar Storage

The Rapid engine is ShannonBase's in-house secondary storage engine. It uses a three-level in-memory columnar structure — Table → IMCU → CU — together with batch (vectorized) execution, ART indexes and Storage Index, maximizing analytical throughput while preserving consistency with InnoDB transactions.

Copyright © 2026, ShannonBase Project Contributors / Public

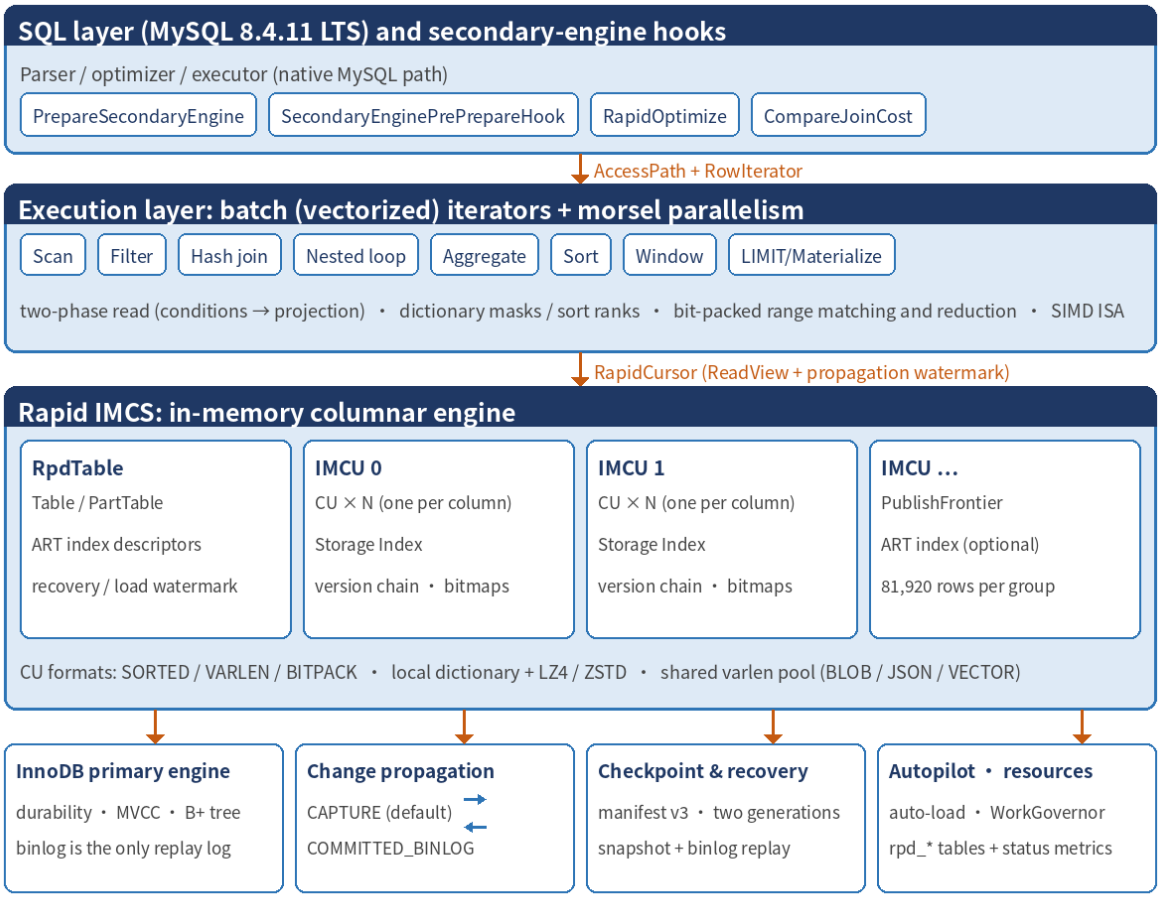


Figure: Layered architecture of the Rapid engine

Three-Level Storage Architecture

Rapid organizes data into three levels:

- Table (RpdTable): manages table-level metadata, one ART index per MySQL key, the Storage Index and the table's IMCU set. Both regular tables (Table) and partitioned tables (PartTable) are supported; partitioned tables are scanned partition by partition in parallel during load, and a load watermark prevents changes already included in the scan result from being applied twice.
- IMCU (In-Memory Column Unit): a logical row-grouping unit, one IMCU per 81,920 rows by default. Each IMCU maintains a delete bitmap, per-column NULL bitmaps, a lightweight transaction log (transaction ID/SCN/operation type), per-column Storage Index and a row directory; read/write visibility is guaranteed by the publisher (PublishFrontier) — only the prefix of contiguously written rows is visible to readers.

Copyright © 2026, ShannonBase Project Contributors / Public

- CU level (column unit): each column in an IMCU maps to one CU, with slot width equal to the normalized internal representation width (4-byte dictionary IDs for string columns). Column encoding is declared in the column comment: RAPID_COLUMN=ENCODING=SORTED|VARLEN|BITPACK; when undeclared, plain slots with LZ4/ZSTD block compression are used (left uncompressed when the gain is below 10%). String columns use local dictionary encoding, and variable-length values such as BLOB/TEXT/JSON/VECTOR live in a shared variable-length pool.
- **ART index: each MySQL key is compiled into one Adaptive Radix Tree. The key encoding depends on the column type: KEY_IMAGE (equality only), SORTABLE_VALUE (order-preserving for numeric/decimal) or COLLATION_WEIGHT (order-preserving for strings by collation weight). A range scan is announced only when every key part is order-preserving; otherwise the index serves point lookups only. Index concurrency is protected by tree-level read/write locks and a version number, and cursors reposition by key when the version changes.**
- **Bit-packed columns (BITPACK): integer columns can be stored with 1–32-bit encoding over their value range, supporting range matching directly on the packed stream, batch decoding and aggregate reduction (COUNT/SUM/MIN/MAX reduce over the encoding without materializing the column). Packed streams can be persisted with checkpoints; when the value range exceeds 32 bits, the column keeps plain slots.**

Storage Index

Every IMCU automatically maintains column-level zone maps (minimum, maximum, NULL count, estimated NDV), a technique similar to Oracle's Storage Index.

During execution the optimizer compares predicates against these IMCU statistics and skips IMCUs that cannot match;

Table-level ColumnStatistics additionally provide equi-height histograms, HyperLogLog NDV, pool sampling and quantiles for selectivity and cost estimation.

Any UPDATE/DELETE invalidates the affected zone maps (invalidate_pruning), so that pruning always stays conservative;

MVCC and Transactional Consistency

Copyright © 2026, ShannonBase Project Contributors / Public

Rapid MVCC has two parts: the IMCU lightweight transaction log records transaction identity and commit SCN, and the CU column-level version chain keeps pre-UPDATE column data. Visibility is decided with the same InnoDB ReadView as the source statement (plus the watermark of the committed-binlog view) rather than by SCN comparison alone; uncommitted transactions are visible to no reader. GC removes unreachable old versions by the minimum active SCN; deleted rows are marked in the delete bitmap, and the O(1) fast path is taken when the whole IMCU is visible.

Vectorized Execution and SIMD Acceleration

Rapid uses a batch (vectorized) execution pipeline: scan, filter, hash join, nested loop, aggregate, sort, window, LIMIT, materialization and result adapters all process data in column chunks (ColumnChunk), keeping expressions and joins on the batch path wherever possible. Dictionary-encoded columns support equality/inequality and LIKE matching directly on encoded IDs, and bit-packed columns support range matching and reduction on the encoded stream; SIMD kernels are dispatched to x86 and ARM through a unified ISA abstraction layer.

Two-phase read: the filter operator first requests only the condition columns and keeps the matching physical row numbers; once the selection is final, the projection columns are fetched, avoiding full-column reads for filtered-out rows. Fetching and condition reads share the same scan context and ReadView, and the cursor may not be repositioned between them.

Parallel Query Execution (Morsel)

Eligible scan, filter and hash-join pipelines run in parallel by morsel: ParallelQueue claims ranges of 4,096 rows dynamically from a single atomic cursor, avoiding the long tail caused by static partitioning; worker threads use private internal THDs, private column buffers and private expression buffers, and only read the coordinator's cursor pin, predicates and ReadView.

Whether to parallelize is decided from the number of rows that can actually participate rather than from optimizer estimates, and slots are reserved through WorkGovernor: rapid_parallel_workers is the session-level cap (default 1) and rapid_parallel_max_workers the process-level cap (read-only at start-up).

Checkpoints and Recovery

With rapid_reload_on_restart enabled, checkpoints are produced per table in the background:

Copyright © 2026, ShannonBase Project Contributors / Public

a snapshot plus manifest v3 (table identity, schema fingerprint, SCN, committed-binlog boundary and per-IMCU checksums), with two generations retained by default. Start-up performs asynchronous recovery first: validate and load the checkpoint, then replay the committed binlog after the recorded boundary; if any step fails, the image is discarded and the table is reloaded from InnoDB.

Checkpoint creation requires no pending notifications and no uncommitted versions, and a generation that fails final validation is discarded wholesale. Until recovery completes, the table does not appear in performance_schema.rpd_table_id.

Resource Governance and Observability

Query and operator memory is managed through grants. Sort, hash, window and materialization spill automatically when they exceed their grant;

rapid_resource_isolation supports five modes — OFF/OBSERVE/SOFT/STRICT_CPU/AUTO — so that analytical, propagation and maintenance work can be placed in different resource groups.

Observability covers six performance_schema.rpd_* tables, 13 status metrics, and per-statement RAPID_STATS operator counters in debug builds.

Parallel Load

Large tables are loaded into the Rapid engine through multi-threaded parallel scans over InnoDB partitions (or over shards of non-partitioned tables).

Each worker clones an independent handler cursor with its own THD and TABLE context and shares only TABLE_SHARE metadata; parallelism is controlled by rapid_parallel_load_threshold / rapid_parallel_part_load_threshold, and load falls back to serial mode when resource governance is in a restricted mode.

The load scan runs at READ COMMITTED and reads committed data only; when it finishes, statistics are frozen, the Storage Index is refreshed and BITPACK columns are packed.

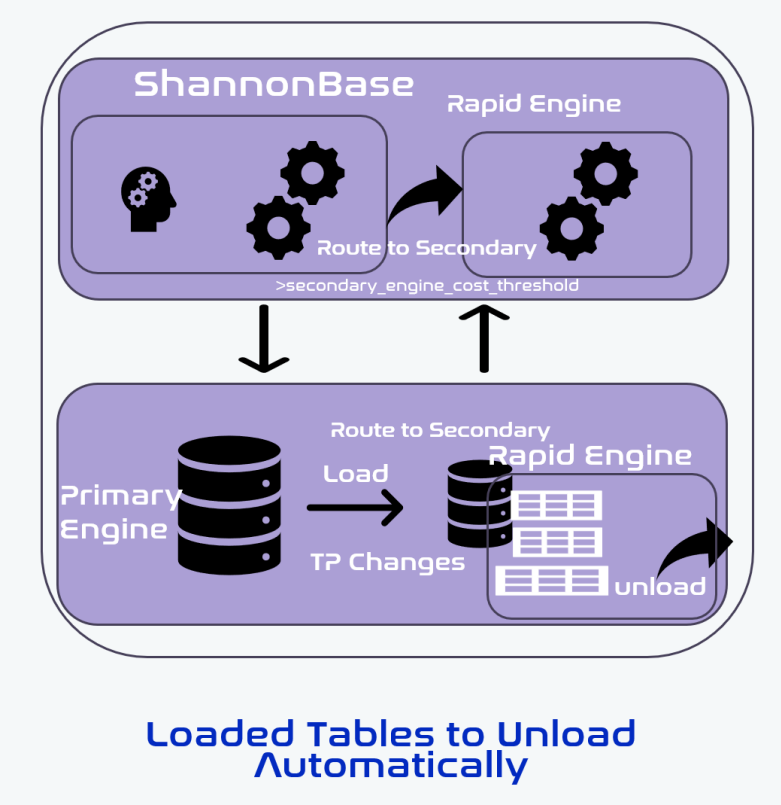
Autopilot — The Self-Managing Memory Engine

ShannonBase Autopilot is an autonomous memory-management subsystem built into the Rapid engine. It continuously monitors query workload, scores every candidate table by access importance and decides automatically which tables to load into or evict from Rapid — with no DBA

Copyright © 2026, ShannonBase Project Contributors / Public

involvement. Its guiding principle is simple: the Rapid memory pool should always hold the most frequently used data.

```
mysql> show variables like "rapid_self_load%";
+-----------------------------------------------------------+---------+
| Variable_name                                             | Value   |
+-----------------------------------------------------------+---------+
| rapid_self_load_base_relation_fill_percentage | 70      |
| rapid_self_load_enabled                                   | OFF     |
| rapid_self_load_interval_seconds                          | 86400 |
| rapid_self_load_skip_quiet_check                          | OFF     |
+-----------------------------------------------------------+-------+
4 rows in set (0.01 sec)
```



Traditional columnar acceleration systems require administrators to specify which tables to

load, which leads to stale configuration as workloads evolve, wasted memory on cold data and load failures when data outgrows the capacity estimate. Autopilot removes all of these failure modes through continuous, workload-driven automation.

Table Importance Scoring

At the core of Autopilot's decision making is a per-table importance score that is updated after every SQL query. When a query completes, the system identifies all tables it touched and updates each table's importance score with the following model:

```
importance_delta = (table_size / total_query_size) × query_execution_time × engine_weight
new_importance = old_importance × (1 − α) + importance_delta × α
```

Two design choices make the model more robust in practice. First, the engine-weight factor distinguishes queries routed to the InnoDB primary engine (weight = 1.0, indicating that they would have benefited from Rapid acceleration) from queries already executed on the Rapid secondary engine (weight = 0.5, because their shorter execution time would otherwise understate their importance). Second, an exponentially weighted moving average (EWMA) prevents brief access spikes from causing unnecessary churn in the memory layout.

Time-Based Importance Decay

Autopilot applies exponential time decay to all importance scores in every scheduling cycle. The decay factor is applied according to the number of days since the table was last queried:

```
decayed_importance = current_importance × DECAY_FACTOR ^ days_since_last_query
```

Once a score decays below the minimum threshold it is set to zero, marking the table as a cold-data candidate. Tables whose importance has dropped to zero and that have not been queried for longer than COLD_TABLE_DAYS are evicted automatically from Rapid memory during the cold-table eviction phase of the next cycle, freeing memory for hotter data.

Copyright © 2026, ShannonBase Project Contributors / Public

Four-Phase Auto-Loading Algorithm

In each scheduling cycle Autopilot executes four phases in order:

1. Importance decay: apply time-based exponential decay to all tracked tables, reflecting the natural decline of access relevance over time.
2. Cold-table eviction: identify self-managed tables whose importance has fallen to zero and whose last query time exceeds the cold-data threshold, and evict them proactively before making load decisions, reclaiming memory.
3. Priority-queue construction: build a max-priority load queue (unloaded tables with importance > 0, in descending importance) and a min-priority eviction queue (loaded self-managed tables that are replacement candidates, in ascending importance).
4. Memory-aware load/evict execution: take the highest-importance candidate from the load queue. If current Rapid memory usage plus the candidate's estimated size stays within the configured memory threshold, load it immediately; if memory is insufficient, evict the least important table from the eviction queue first and retry. The process continues until the load queue is exhausted or the memory threshold is fully consumed.

Memory Threshold Management

Autopilot enforces a configurable fill cap on the Rapid memory pool:

```
memory_threshold = rapid_memory_size_max × self_load_base_relation_fill_percentage ÷ 100
```

The default fill ratio is 70%, reserving 30% of the Rapid memory pool as a buffer for user-initiated manual loads, change-propagation write buffers and temporary query execution memory. This headroom prevents Autopilot's background activity from competing with the working memory that foreground analytical queries require.

System Quiet Detection

To keep background loading from competing with peak-period OLTP and HTAP traffic, Autopilot performs a system quiet check before each scheduling cycle. The system is considered active if any monitored table was queried within the last QUERY_QUIET_MINUTES minutes, or if

Copyright © 2026, ShannonBase Project Contributors / Public

change propagation from InnoDB to Rapid is still in progress for any table. If the system is active, Autopilot waits at a configurable interval (QUIET_WAIT_SECONDS) for up to MAX_QUIET_WAIT_ATTEMPTS attempts and then skips the cycle entirely, waiting for the next scheduling interval.

For offline or batch-oriented deployments, the mechanism can be bypassed by setting rapid_self_load_skip_quiet_check = ON.

User-Load Protection

Autopilot distinguishes strictly between two load origins — USER (explicitly loaded by an administrator through ALTER TABLE ... SECONDARY_LOAD) and SELF (managed automatically by Autopilot).

When the eviction algorithm selects a USER-loaded table as a replacement candidate, Autopilot does not evict it. Instead it downgrades the table's load type to SELF while keeping the table resident, and writes a warning to the log to notify the DBA. Fully evicting a user-loaded table requires an explicit SECONDARY_UNLOAD command. This protection ensures that a DBA's deliberate load decisions are never silently overridden.

Background Coordinator Thread

Autopilot runs as a dedicated background thread (self_load_coordinator) with full lifecycle management. The scheduling interval is controlled by rapid_self_load_interval_seconds (default: 86400 seconds / 24 hours), letting administrators place Autopilot cycles in off-peak windows. Thread lifecycle transitions are atomic and coordinated through condition variables, ensuring clean start-up and shutdown without blocking the MySQL server's foreground thread pool.

Metadata Bootstrap

At start-up Autopilot scans four system catalog tables to build a complete view of all candidate tables, with no external configuration:

System table scanned	Information extracted
mysql.schemata	Schema ID → schema name mapping for all user databases

Copyright © 2026, ShannonBase Project Contributors / Public

mysql.table_stats	Row count, data size, index size → estimate each table's memory footprint
information_schema (DD API)	InnoDB internal table ID per table (se_private_id)
mysql.tables	Primary engine = InnoDB filter; SECONDARY_ENGINE = RAPID flag; partition detection

After bootstrap, the mirror registry (RpdMirror::Registry, performance_schema.rpd_mirror) is kept current through DDL event hooks: CREATE TABLE adds an entry, DROP TABLE removes it, and ALTER TABLE updates the secondary-engine assignment state.

Autopilot Capabilities at a Glance

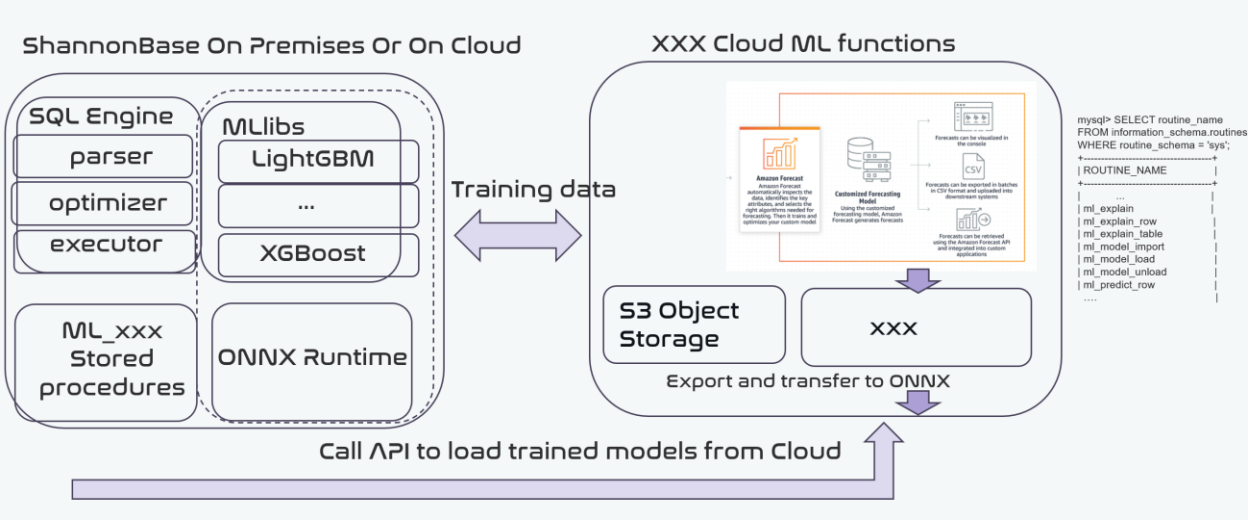
Autopilot capability	Mechanism
Hot-table detection	Per-query importance score: table size share × execution time × engine weight, with EWMA smoothing
Cold-data eviction	Exponential time decay + cold-data age threshold → automatic eviction from Rapid memory
Memory-aware load decisions	Fill-ratio cap; a memory pre-check precedes every table load
Priority-driven memory replacement	Max-heap load queue + min-heap eviction queue; the least important table is replaced first
No interference with OLTP	Quiet detection: skip the cycle if any table was queried in the last N minutes or propagation is still running
User-Load Protection	USER tables are downgraded to SELF on eviction attempts; never removed silently
Schema awareness	Bootstrap from mysql.* system tables at start-up; DDL event hooks keep it consistent in real time
Configurable	Interval, fill ratio, quiet detection and quiet wait are all configurable through

Copyright © 2026, ShannonBase Project Contributors / Public

scheduling	system variables

AutoML — In-Database Machine Learning

ShannonBase AutoML provides a fully integrated, SQL-accessible machine learning pipeline that performs model training, prediction generation and explainability analysis inside the database engine — with no extraction of data to an external ML platform, no Python dependency for end users, and no additional cost beyond the database instance itself.



The Challenge of Machine Learning on Database Data

The traditional approach to machine learning on database data usually requires users to:

- Export training data from the OLTP database through ETL
- Load the data into a separate data-science environment (Jupyter, Spark, SageMaker)
- Choose algorithms, engineer features, tune hyperparameters and validate models
- Deploy the trained model to a separate inference service
- Write custom integration code so that application SQL can call that inference service

This multi-step process brings data latency, multiplied infrastructure cost and security exposure, and it requires deep machine-learning expertise that most database teams do not have.

How AutoML Works

ShannonBase AutoML eliminates all of these steps. Training data comes directly from InnoDB

Copyright © 2026, ShannonBase Project Contributors / Public

or Rapid tables. The entire machine learning pipeline — data preprocessing, algorithm selection, feature selection, hyperparameter optimization, explainer training and model persistence — runs inside the database engine against the live dataset.

Trained models are stored in the user schema (ML_SCHEMA_<username>.MODEL_CATALOG) and participate in standard database backup, restore and access control.

Capability	Description
Fully automated pipeline	Algorithm selection, feature engineering, hyperparameter tuning and explainer training are automated end to end
SQL interface	sys.ML_TRAIN / ML_MODEL_LOAD / ML_MODEL_UNLOAD / ML_MODEL_IMPORT / ML_MODEL_EXPORT / ML_MODEL_ACTIVE / ML_SCORE / ML_PREDICT_TABLE / ML_EXPLAIN / ML_EXPLAIN_TABLE, plus the native ML_MODEL_* functions
Supported task types	Classification, regression, time-series forecasting, anomaly detection (PCA/GLOF/GKNN), recommendation, topic modeling
Core engines	LightGBM for training; ONNX Runtime for inference and external ONNX model import
Explainability	Feature-attribution scores for every prediction, produced by integrated explainer models
Model metadata	MODEL_CATALOG records task, target column, training table, column names, model quality and training drift metrics
Security	Models and data stay inside the instance by default; prompts reach a cloud service only when a provider is configured explicitly
Cost	No additional charge; delivered with the ShannonBase instance

Usage Example

A classification model can be trained and used for predictions with standard SQL only:

Copyright © 2026, ShannonBase Project Contributors / Public

```
-- Train the model
SET @model = 'census_model';
CALL sys.ML_TRAIN('shannon_ml.census_train', 'revenue',
                JSON_OBJECT('task','classification'), @model);

-- Load the model and score it
CALL sys.ML_MODEL_LOAD(@model, NULL);
CALL sys.ML_SCORE('shannon_ml.census_test', 'revenue', @model,
                'balanced_accuracy', @score, NULL);

-- Row-level and table-level prediction
SELECT sys.ML_PREDICT_ROW(JSON_OBJECT('age',41,'education','Bachelors'),
                        @model, NULL);
CALL sys.ML_PREDICT_TABLE('shannon_ml.census_test', @model,
                        'shannon_ml.census_pred', NULL);
```

Training and Inference Performance

AutoML training throughput depends on data volume and feature dimensionality. Because training data is read directly from InnoDB or Rapid, the export and load steps disappear; algorithm selection, feature engineering, hyperparameter search and explainer training all happen inside the instance.

For online inference, ML_PREDICT_ROW() caches the loaded booster on the batch prediction path, avoiding repeated model loads; single-row inference usually completes in milliseconds and can be combined with transactional queries.

Vector Search and Generative AI

ShannonBase natively integrates vector data storage, similarity search, in-database embedding and local LLM inference. All of these capabilities are built into the database kernel, so retrieval-augmented generation (RAG) and semantic search applications can be built entirely inside the database without deploying a separate vector database or external LLM APIs.

Native VECTOR Data Type

ShannonBase introduces a native VECTOR type into the MySQL type system and provides the TO_VECTOR / STRING_TO_VECTOR, FROM_VECTOR / VECTOR_TO_STRING, VECTOR_DIM and

DISTANCE(v1, v2, 'COSINE'|'DOT'|'EUCLIDEAN'|'MANHATTAN') functions.

Columns of the vector type are evaluated with batch scans; similarity search is implemented as a distance expression over columnar scans.

```
CREATE TABLE embeddings (
   id INT PRIMARY KEY,
   description TEXT,
   embedding VECTOR(10)) secondary_engine=rapid;


INSERT INTO embeddings (id, description, embedding)
VALUES (1, 'Example text', TO_VECTOR("[0.12, -0.34, 0.56, 0.78, -0.91, 0.23, -0.45, 0.67, -0.89, 1.23]"));


SELECT LENGTH(embedding), FROM_VECTOR(embedding) FROM embeddings WHERE id = 1;
SELECT DISTANCE(STRING_TO_VECTOR("[1.01231, 2.0123123, 3.0123123, 4.01231231]"),
STRING_TO_VECTOR("[1, 2, 3, 4]"), "COSINE");
SELECT DISTANCE(STRING_TO_VECTOR("[1.01231, 2.0123123, 3.0123123, 4.01231231]"),
STRING_TO_VECTOR("[1, 2, 3, 4]"), "DOT");
SELECT DISTANCE(STRING_TO_VECTOR("[1.01231, 2.0123123, 3.0123123, 4.01231231]"),
STRING_TO_VECTOR("[1, 2, 3, 4]"), "EUCLIDEAN");
```

In-Database Embedding Generation

sys.ML_EMBED_ROW and sys.ML_EMBED_TABLE run embedding inference inside the database process with ONNX Runtime. model_id is the model name (for example multilingual-e5-small, all-MiniLM-L12-v2); models live in <basedir>/llm-models by default and the location can be set with --llm-dir. Text never leaves the database process while an embedding is computed.

```
SELECT sys.ML_EMBED_ROW('What is artificial intelligence?', JSON_OBJECT('model_id', 'all-MiniLM-L12-v2'));
CALL sys.ML_EMBED_TABLE('sh_embedding.docs.body', 'sh_embedding.docs.embedding',
JSON_OBJECT('model_id', 'all-MiniLM-L12-v2', 'batch_size', 500));
```

Copyright © 2026, ShannonBase Project Contributors / Public

In-Database LLM Inference

ShannonBase supports LLM inference inside the database process through ML_GENERATE(): the local backend is ONNX Runtime (KV cache, CPU/GPU chosen by host probing, no user intervention needed), with Ollama as an alternative, or a MaaS cloud service through model_options.provider (deepseek / dashscope / anthropic / qianfan).

Local ONNX models (such as Qwen3.5-2B-ONNX) need no external API; when a MaaS cloud service is selected, the prompt is sent to that service and must be configured explicitly by the user. (This may expose sensitive data contained in the context.)

```
SELECT sys.ML_GENERATE(
   "What is ShannonBase, gvie me some explaination in common?",
   JSON_OBJECT(
      "task", "generation",
       "model_id", "Qwen3.5-2B-ONNX",
      "language", "en",
      "temperature", 0.8,
       "top_p", 0.9,
       "repeat_penalty", 1.1,
      "max_tokens", 256
   )
);
SELECT sys.ml_generate(
       "Contact John Doe at john.doe@example.com or call 555-867-5309.",
       JSON_OBJECT("task", "pii_detect", "model_id", "multilang-pii-ner-ONNX", "output_format", "json")
     ) AS pii_report;
```

See the related documentation for further details.

RAG Architecture

Copyright © 2026, ShannonBase Project Contributors / Public

A complete retrieval-augmented generation pipeline can run the whole RAG and inference process through ShannonBase SQL:

1. Store source documents as rows in a table
2. Generate and store embedding vectors with ML_EMBED()
3. Perform similarity search with the VECTOR distance functions in SQL
4. Combine the retrieved context with ML_GENERATE() to produce a grounded LLM response

```
INSERT INTO demo_embeddings (segment, text, document_name, segment_number, metadata)
VALUES
('What is AutoML?', ('AutoML explanation example text'), 'doc1', 1, JSON_OBJECT('category', 'AI')),
('AutoML automates model building', ('AutoML automates model building'), 'doc1', 2,
JSON_OBJECT('category', 'AI')),
('Machine learning basics', ('Basics of machine learning'), 'doc2', 1, JSON_OBJECT('category', 'ML')),
('ShannonBase is an AI-native SQL engine', ('ShannonBase overview'), 'doc3', 1, JSON_OBJECT('category',
'DB'));

CALL sys.ML_EMBED_TABLE('sh_rag.demo_embeddings.text', 'sh_rag.demo_embeddings.embedding',
JSON_OBJECT('model_id','all-MiniLM-L12-v2','batch_size',500));

SET @options = JSON_OBJECT(
   'vector_store', JSON_ARRAY('sh_rag.demo_embeddings'),
   'n_citations', 2,
   'embed_model_id', 'all-MiniLM-L12-v2',
   'vector_store_columns', JSON_OBJECT(
      'segment', 'segment',
      'segment_embedding', 'embedding',
      'document_name', 'document_name',
      'metadata', 'metadata',
      'segment_number', 'segment_number'
```

```
    )
);
```

Because all four steps execute inside the same database engine, there is no inter-process communication, no data serialization overhead and no external API rate limiting. The whole RAG pipeline benefits from Rapid's vectorized execution and parallel scans.

JavaScript Multilingual Engine (MLE)

ShannonBase implements in-database JavaScript execution through an embedded JerryScript engine: users can write stored procedures and functions in JavaScript, execute SQL inside them and emit result sets. This meets the need to keep complex business logic, data preprocessing, feature engineering and AI post-processing close to the data.

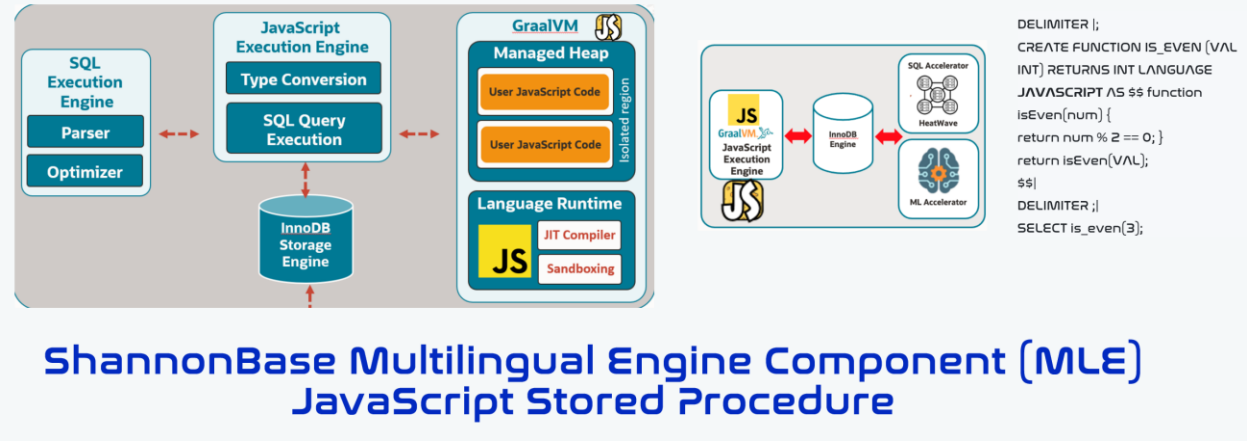
Dimension	JavaScript MLE vs. C/C++ UDF
Development cost	Low (JavaScript) vs. high (C/C++, requires MySQL kernel expertise)
Security	High — sandboxed execution environment vs. low — direct process-level access
Hot reload	Modify and rebuild the routine and it takes effect (no recompilation or server restart) vs. reloading a server plugin
Complex logic	Native strings/JSON/regular expressions vs. manual implementation
AI integration	Direct access to ML_EMBED, ML_GENERATE and vector arithmetic vs. custom C bindings

JavaScript MLE participates in ShannonBase's unified execution context. JavaScript functions can access InnoDB transactional data, issue Rapid analytical queries, call ML_EMBED() and ML_GENERATE(), execute SQL through exec_sql() and return result sets directly to the client through send_result_set().

Copyright © 2026, ShannonBase Project Contributors / Public

```
DELIMITER |;
CREATE FUNCTION IS_EVEN (VAL INT) RETURNS INT
LANGUAGE JAVASCRIPT AS $$
function isEven(num) {
    return num % 2 == 0;
}
return isEven(VAL);
$$|
DELIMITER ;|

SELECT is_even(3);
```



ShannonBase Multilingual Engine Component (MLE) JavaScript Stored Procedure

Native-Agent Inside the Database

ShannonBase Native-Agent is an AI-native conversational agent embedded directly in the ShannonBase database kernel. Through a single, SQL-accessible natural-language interface it exposes the full capability of the database — analytical queries, operational diagnostics, data changes, knowledge retrieval and multi-step reasoning — with no external middleware, no additional deployment and no application-layer changes.

The agent is triggered by a single SQL function call that any MySQL-compatible client can issue:

```
-- Core invocation
SET @chat_options = JSON_OBJECT('conversation_id', UUID());
CALL sys.shannon_chat('<natural language instruction>');  -- user message
```

sys.shannon_chat is a SQL PROCEDURE; results are returned as a result set (containing response and, depending on the path, trace_id and citations) rather than as a scalar return value aliased with AS. Session configuration (conversation ID, model and RAG options, approval policy and so on) is passed through the @chat_options session variable.

```
-- Example
SET @chat_options = JSON_OBJECT('conversation_id', 'session-dba-20240601');
CALL sys.shannon_chat('Show me the top 5 tables by row count in the current database.');
```

Design Motivation: Closing the Gap Between Data and Intelligence

Traditional databases require users to know SQL, understand the schema and write queries by hand. As databases begin to host machine-learning models, vector embeddings and in-process LLM inference, their operational surface expands rapidly.

DBAs, data analysts and application developers must master an ever-growing set of SQL primitives — ML_TRAIN, ML_GENERATE, ML_EMBED, vector distance functions, system status tables — merely to extract value from the data platform.

Native-Agent closes that gap. By embedding a reasoning agent directly in the kernel, ShannonBase lets any user interact with the entire data platform in natural language through a single SQL call — HTAP queries, AutoML training, RAG retrieval, system diagnostics and transactional changes.

Query-execution expertise, schema awareness and model-invocation logic all live inside the database rather than in application code. Users can of course also use the plugin mechanism to write agents in JavaScript that fit their own business logic.

Copyright © 2026, ShannonBase Project Contributors / Public

Architecture

Native-Agent is implemented as a layered pipeline that runs entirely inside the ShannonBase process. The table below shows the internal request lifecycle from a natural-language user message to a reliable SQL-grounded response.

Layer	Component	Role
Entry point	sys.shannon_chat(msg) — PROCEDURE	SQL PROCEDURE — the only public interface, invoked with CALL. Accepts the natural-language message; the conversation ID and remaining configuration are passed through the @chat_options session variable, and the result is returned as a result set (response and, depending on the path, trace_id and citations).
Plugin dispatch	Plugin discovery (L1–L4)	Checks session-level, registry-level, convention-named and default agent implementations in priority order.
Request router	Intent classifier	Handles pending approvals first; remaining requests are routed to Route A (exact catalog match) or Route D (LLM agent loop).
Schema retrieval	ML_RETRIEVE_SCHEMA_METADATA	Semantic similarity search over prebuilt schema embeddings, identifying the relevant tables and columns without injecting the full DDL.
SQL generation	LLM (ML_GENERATE)	Assembles and executes SQL against InnoDB or Rapid. Validates syntax and retries with error-correction prompts when

Copyright © 2026, ShannonBase Project Contributors / Public

Layer	Component	Role
		execution fails.
Knowledge retrieval	ML_RAG / ML_EMBED	Vector similarity search over user-indexed knowledge tables, used to answer knowledge questions.
Memory manager	mysql.agent_memory	Persists every turn (role, content, chain of thought) keyed by conversation_id. Turns are loaded to build the multi-turn context window.
Write safety	Transaction Guard	Enforces begin_tx before any DML. Validates that UPDATE/DELETE statements carry a WHERE clause. Rolls back automatically when the agent fails.
Response assembly	Result formatter	Combines SQL result sets, RAG snippets and LLM-generated text into a coherent natural-language answer for the caller.

Intent Routing and Execution Paths

The request router first checks for pending approvals, then classifies the message into one of two execution paths. Classification happens before any LLM call, providing a fast path for frequent operational queries:

Execution path	Trigger	Mechanism	LLM call
A: exact catalog match	catalog_match() matches a schema/object name	Executes a preset SQL statement through the unified tool guard; no LLM call	No
D: LLM agent loop	All other requests	Tool-calling loop (up to 10 rounds): schema/knowledge context → LLM	Yes

Copyright © 2026, ShannonBase Project Contributors / Public

Execution path	Trigger	Mechanism	LLM call
		→ tool → safety net	
Approval resumption	Message contains Approve / Reject / Modify:SQL	Resumes the approval state machine and advances the plan step by step (CAS-protected)	No (approved steps only)

Catalog fast-path queries bypass the LLM entirely and return in milliseconds by querying known system views directly. This is especially valuable for DBA monitoring workflows where response latency is critical.

Conversation Memory and Multi-Turn Context

Native-Agent persists conversation state in the mysql.agent_memory system table. Each turn is stored as one row holding the role (user or assistant), the message content, the internal reasoning chain (thought) and a creation timestamp, keyed by the user-supplied conversation_id; companion tables — agent_memory_audit, agent_sql_trace, agent_review_history, agent_tx_lease, agent_policy and shannon_agent_plugins — record audits, approvals, transaction leases and plugin registrations.

On every turn the agent loads the most recent N turns from agent_memory (default: 5, corresponding to memory_options.short_term.recent_turns and configurable through chat_options) to build the context window passed to the LLM. This makes coherent multi-turn conversations possible, in which follow-up questions refer to earlier results in natural language.

```
SET @chat_options = JSON_OBJECT('conversation_id', UUID());

CALL sys.shannon_chat('How many rows are in the orders table?');          -- q1
CALL sys.shannon_chat('Break that down by status.');                      -- q2
CALL sys.shannon_chat('Which status had the highest average value?');     -- q3
CALL sys.shannon_chat('Now show me the monthly trend for that status.');  -- q4
```

The agent resolves "that status" in Q4 back to the Q3 result, with no application-level state management at all. All context passing is handled transparently by the memory subsystem.

Parameter	Type	Default	Role
history_length	INT	5	Number of previous turns included in the LLM context window
schema_name	STRING	current DATABASE()	Restricts schema vector search to the given schema
retrieve_top_k	INT	6	Number of RAG snippets retrieved per knowledge query
skip_generate	BOOL	false	When true, skip LLM generation when RAG retrieval returns nothing
model_options	JSON	temperature=0.25, max_tokens=1200	LLM generation parameters: temperature, top_p, max_tokens, repeat_penalty
conversation_id	STRING	(continues the previous conversation)	Conversation UUID. If omitted, the connection continues its most recent conversation; set to :new to force a new one
model_options.provider	VARCHAR	(local ONNX / Ollama)	Selects a remote LLM provider: deepseek | dashscope | anthropic | qianfan
review_mode	STRING	off	When set to review, writes / DDL / high-risk SQL pause for human approval (see "Transactional Write

Copyright © 2026, ShannonBase Project Contributors / Public

Parameter	Type	Default	Role
			Safety")

Schema-Aware SQL Generation

Accurate SQL generation requires the agent to resolve natural-language references to the right table and column names. For databases with few tables the agent injects the full DDL into the LLM prompt. For larger schemas (roughly more than 20 tables) this degrades prompt quality and increases latency.

To solve this, Native-Agent works with the ShannonBase vector infrastructure: after schema embeddings are prebuilt with ML_RETRIEVE_SCHEMA_METADATA, the agent performs semantic similarity search over the embedded schema catalog and fetches only the most relevant table definitions for each query. This shortens prompts, improves SQL accuracy and sharply reduces token consumption for complex schemas:

```
-- Build the schema vector index (one-off; about 1-3 minutes for a large schema)
CALL sys.ML_RETRIEVE_SCHEMA_METADATA('',@result,
   JSON_OBJECT('schemas', JSON_ARRAY(DATABASE()), 'build_index', true));
-- Verify readiness
SELECT COUNT(*) AS indexed_tables
FROM mysql.schema_embeddings
WHERE schema_name = DATABASE() AND status = 1;
```

Schema embeddings must be rebuilt after DDL changes (ALTER TABLE, CREATE TABLE, DROP TABLE).

Embeddings are stored in mysql.schema_embeddings and participate in the standard database backup and restore process.

Transactional Write Safety

Native-Agent supports natural-language-driven data changes (INSERT, UPDATE,

Copyright © 2026, ShannonBase Project Contributors / Public

DELETE) inside a structured transactional safety framework. Writes are constrained by four invariants enforced by the Transaction Guard:

Safety mechanism	Description
Mandatory transaction wrapping	All DML statements are issued inside an explicit begin_tx / commit_tx block managed by the agent. Writes outside an active transaction are rejected.
Mandatory WHERE-clause validation	UPDATE and DELETE statements without a WHERE predicate are rejected before execution, preventing accidental full-table changes.
Rollback on failure	If the agent loop exits abnormally at any point in a write sequence, any open transaction is forcibly rolled back before the response is returned.
Pre-execution confirmation	With review_mode="review" in @chat_options, writes / DDL / high-risk SQL pause and produce a structured approval prompt (plan ID, SQL, affected tables, estimated rows, risk level); execution continues only after the user replies Approve / Reject / Modify:<SQL>.

The user always keeps full control: issuing a rollback instruction at any point in a write sequence makes the agent emit ROLLBACK and abandon the operation.

Writes inherit that user's MySQL privilege boundary; the agent executes with the same privileges as the session user.

Plugin Extension Model

Native-Agent supports a four-level plugin discovery hierarchy, letting users and operators inject custom business logic into the agent's processing chain without modifying the kernel. Plugin dispatch happens before the default agent engine is invoked:

Copyright © 2026, ShannonBase Project Contributors / Public

Priority	Mechanism	Scope	Configuration
L1 (highest)	Session variable	Current session only	SET @shannon_agent_plugin = 'schema.function'
L2	Plugin registry	Instance-wide; multiple plugins supported	CALL sys.shannon_register_plugin(name, schema, func, priority, desc, @result)
L3	Convention naming	Per schema, zero configuration	CREATE FUNCTION shannon_agent(msg TEXT, conv_id VARCHAR(64)) RETURNS TEXT LANGUAGE JAVASCRIPT
L4 (fallback)	System default	Always available	No configuration needed

Plugins are JavaScript stored procedures executed in the ShannonBase MLE environment. They receive the raw user message and the conversation ID, and can replace the default agent with business-specific context (such as currency units, time-zone rules or access policies) before delegating to the default agent (sys.shannon_agent_default).

This architecture ensures that domain-specific customization never requires forking the core agent logic:

```
-- Plugin example: inject business context before calling the default engine
CREATE FUNCTION shannon_agent(
    user_message    TEXT,
    conversation_id VARCHAR(64)
) RETURNS TEXT
SQL SECURITY INVOKER LANGUAGE JAVASCRIPT AS $$
function my_agent(msg, conv_id) {
  function esc(s) { return String(s==null?'':s).replace(/\\/g,'\\\\').replace(/'/g,"\\'"); }
```

Copyright © 2026, ShannonBase Project Contributors / Public

```
 function query(sql) { try { return JSON.parse(sys.exec_sql(sql)); } catch(e) { return {}; } }
 var enriched = '[Rules] currency_unit=cents; timezone=UTC+8\n' + msg;
 var rows = query("SELECT sys.shannon_agent_default('" + esc(enriched) + "','" + esc(conv_id) + "') AS
r");
 return (Array.isArray(rows) && rows[0]) ? String(rows[0].r || '') : '';
}
return my_agent(user_message, conversation_id);
$$;
```

Direct Access to ML Primitives

For users who need lower-level control, the same ML infrastructure used inside Native-Agent is reachable as SQL functions.

This makes programmatic feature engineering, batch inference pipelines and RAG construction possible without going through the conversational interface:

Function / procedure	Signature	Description
sys.ML_GENERATE	ML_GENERATE(prompt TEXT, options JSON)	Text generation and LLM inference with any loaded ONNX or remote model
sys.ML_EMBED_ROW	ML_EMBED_ROW(text TEXT, options JSON)	Generates an embedding vector for a single text
sys.ML_EMBED_TABLE	ML_EMBED_TABLE(input_table_column, output_table_column, options)	Generates embeddings in bulk for a text column of a table
sys.ML_RAG	ML_RAG(query TEXT, OUT output JSON, options JSON)	Retrieval-augmented generation: vector retrieval + grounded

Copyright © 2026, ShannonBase Project Contributors / Public

Function / procedure	Signature	Description
		LLM response
sys.ML_RETRIEVE_SCHEMA_METADATA	ML_RETRIEVE_SCHEMA_METADATA(q, @out, options)	Semantic retrieval over schema metadata; also builds the schema embedding index
DISTANCE / TO_VECTOR / FROM_VECTOR / VECTOR_DIM	DISTANCE(v1, v2, metric)	Vector constructors and distance functions for standard SQL (COSINE/DOT/EUCLIDEAN/MANHATTAN)

Integration with the ShannonBase Architecture

Native-Agent is not a separate service — it is a first-class citizen of the ShannonBase engine stack, sitting above the ML & AI layer and the MLE (JavaScript) execution environment.

It consumes other subsystems as internal dependencies:

- Rapid engine: analytical queries generated by the agent pass through the HTAP optimizer and are offloaded to the Rapid column engine where beneficial, automatically benefiting from vectorized execution, Storage Index skipping and parallel scans.
- AutoML: the agent can call ML_TRAIN, ML_PREDICT_ROW and ML_EXPLAIN_TABLE inside multi-step reasoning sequences, enabling conversational model training and inference workflows.
- Vector search: the agent's RAG path calls ML_EMBED and similarity functions for knowledge question answering.

Copyright © 2026, ShannonBase Project Contributors / Public

- JavaScript MLE: agent plugins are written as JavaScript stored procedures executed by the JerryScript engine. The agent core itself (shannon_agent_default) is written in JavaScript, so its logic can be reviewed and extended.
- InnoDB: the mysql.agent_memory table behind the conversation memory system is a standard InnoDB table that takes part in ACID transactions, MySQL replication and standard backup procedures.

Typical Use Cases

Use Cases	How Native-Agent solves it
Ad-hoc analytical queries	Analysts state the question in natural language; the agent generates, executes and returns results from the Rapid column engine — no SQL knowledge required.
DBA operational diagnostics	Keywords such as "deadlock", "slow queries" or "buffer pool" trigger catalog fast-path queries that return system health data in milliseconds, with no LLM inference involved.
Multi-step business analysis	Analysts break complex questions down across turns; the agent keeps context between turns and references earlier results to build progressive analysis.
Schema exploration	New team members can ask "what tables exist", "what does the orders table look like" or "how are customers and orders related" to understand the data model conversationally.
Constrained data changes	Operators issue write instructions in natural language; the Transaction Guard enforces the safety invariants while a complete audit trail is kept in agent_memory.
Domain-specific AI assistants	Teams can deploy custom plugins that inject business terminology, access policies or proprietary context, building a domain-adapted AI assistant without kernel changes.

Copyright © 2026, ShannonBase Project Contributors / Public

Use Cases	How Native-Agent solves it
RAG knowledge-base Q&A	After indexing internal documents with ML_EMBED_TABLE, users can ask knowledge questions grounded in proprietary content, with answers generated entirely inside the database boundary.

Security and Data Governance

Because Native-Agent executes entirely inside the ShannonBase process, every guarantee that applies to direct SQL access also applies to the queries the agent generates:

- Privilege enforcement: every SQL statement the agent generates and executes runs under the credentials and privileges of the connected MySQL user. The agent cannot reach tables or operations that user is not authorized for.
- No external transmission by default: embedding, PII detection and local LLM inference all run in-process through ONNX Runtime. Prompts reach a cloud service only when the user explicitly selects a provider (deepseek/dashscope/anthropic/qianfan); that choice is controlled by chat_options and is auditable.
- Write audit trail: every turn — including the full reasoning chain and the executed SQL statements — is persisted to mysql.agent_memory, providing a complete audit log of all agent-driven operations.
- Production hardening: for sensitive databases, custom plugins can add data masking, row-level access filtering or extra audit logging to the agent pipeline without kernel changes.

Native-Agent Capabilities at a Glance

Capability	Mechanism
SQL-accessible interface	A single sys.shannon_chat(msg) PROCEDURE plus the

Copyright © 2026, ShannonBase Project Contributors / Public

Capability	Mechanism
	@chat_options session variable; usable from any MySQL client with no extra driver
Intent classification and routing	A lexical/semantic classifier dispatches to Route A (exact catalog match) or Route D (LLM agent loop), with approval resumption
Schema-aware SQL generation	Similarity search over ML_RETRIEVE_SCHEMA_METADATA embeddings; only the relevant DDL is fetched
Multi-turn conversation memory	mysql.agent_memory InnoDB table; the history window is configurable through chat_options.history_length
Catalog fast-path diagnostics	Pre-mapped system queries bypass the LLM entirely; sub-10 ms responses for monitoring keywords
Transactional Write Safety	Mandatory begin_tx, WHERE-clause validation, rollback on failure, pre-execution scope confirmation
RAG knowledge Q&A	ML_EMBED + VECTOR similarity search + ML_GENERATE for grounded responses, all inside the database
Plugin extensibility	Four discovery levels; JavaScript MLE plugins; zero-configuration convention naming (L3)
HTAP query acceleration	Analytical SQL generated by the agent is routed transparently to the Rapid column engine by the HTAP optimizer
Data sovereignty	All inference runs in-process by default (ONNX Runtime); prompts reach a cloud service only when a provider is configured explicitly; privilege boundaries always follow the connected user

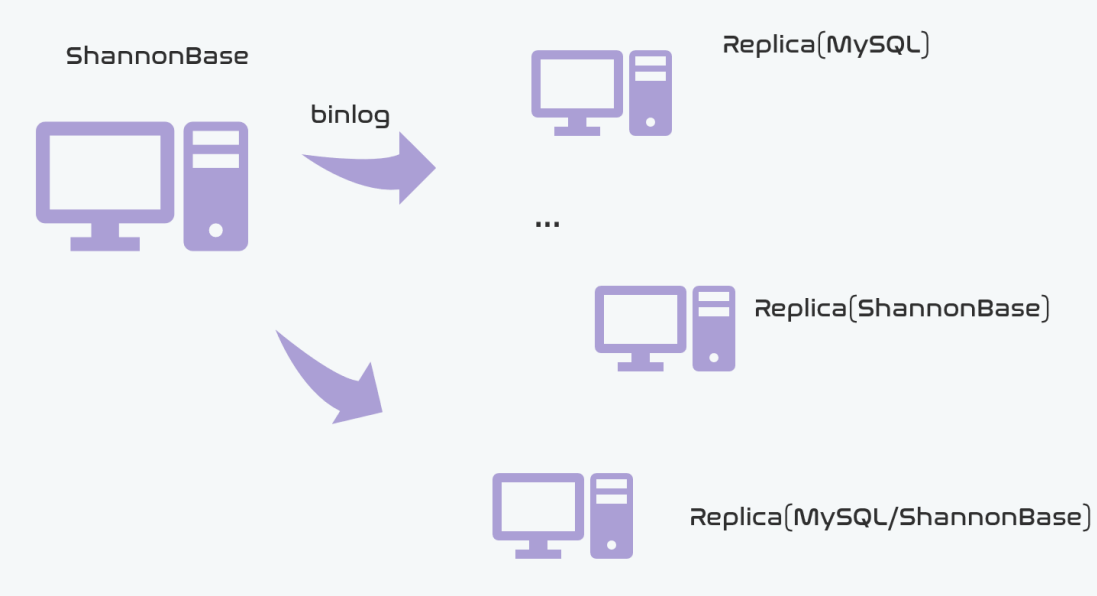
ShannonBase Deployment

Copyright © 2026, ShannonBase Project Contributors / Public

Deployment Models

ShannonBase supports several deployment configurations to fit different organizational needs:

- **Single-node HTAP: InnoDB and Rapid co-located on one server, with Rapid running as an in-process columnar accelerator. Suitable for development, testing and mid-sized production workloads.**
- **Read replica / multi-instance scaling: ShannonBase can serve as a read replica in a MySQL replication topology, receiving upstream binlog and maintaining a local Rapid copy to absorb analytical load; the clustered form that shares Rapid memory across nodes (MPP) is an enterprise-edition capability.**
- **MySQL-compatible hybrid deployment: existing MySQL deployments can add a Rapid-enabled ShannonBase as a read replica, receive binlog-based replication from upstream MySQL and serve analytical queries without affecting OLTP.**



System Variables and Configurability

ShannonBase exposes fine-grained system variables for tuning Rapid memory allocation, parallel load degree, change propagation mode, auto-load scheduling, purge behaviour and GC thresholds. The main categories are:

- Memory and resources: rapid_memory_size_max (upper bound of the in-process Rapid memory pool, default 2G, maximum 64G), rapid_resource_isolation and

Copyright © 2026, ShannonBase Project Contributors / Public

rapid_resource_{ap,sync,maint}_limit, plus the individual spill limits for window/sort/materialize.

- Parallelism: rapid_parallel_workers (session level, default 1), rapid_parallel_max_workers (process level, read-only at start-up), rapid_parallel_load_threshold, rapid_parallel_part_load_threshold and rapid_parallel_load_max.
- Change propagation and recovery: rapid_change_propagation_mode (CAPTURE|COMMITTED_BINLOG), rapid_reload_on_restart and rapid_query_sync_timeout_ms.
- Auto-loading: rapid_self_load_enabled, rapid_self_load_interval_seconds (default 86400), rapid_self_load_skip_quiet_check and rapid_self_load_base_relation_fill_percentage (default 70).
- GC/purge and observability: rapid_gc_interval_scn, rapid_gc_interval_seconds, rapid_purge_batch_size, rapid_min_versions_for_purge, rapid_purge_efficiency_threshold and rapid_max_purger_timeout, used together with performance_schema.rpd_* and the rapid_* status variables.

Security, Observability and Manageability

- **Privilege boundary: all SQL (including agent-generated SQL) executes under the MySQL privileges of the connected user; Rapid introduces no separate authorization model.**
- **Data protection: TLS, encryption at rest, auditing and backup reuse what MySQL/InnoDB provide; Rapid is only a rebuildable in-memory copy, so backups need to cover only InnoDB and binlog.**
- **Inference data flow: embedding, PII detection and local LLM inference run in-process by default; prompts are sent to a cloud service only when a provider is explicitly selected.**
- **Observability: performance_schema provides six tables — rpd_tables, rpd_table_id, rpd_columns, rpd_column_id, rpd_preload_stats and rpd_mirror; status metrics include rapid_mempool_*, rapid_pop_*, rapid_query_*, rapid_bg_* and the**

Copyright © 2026, ShannonBase Project Contributors / Public

rapid_resource_plan renderer; debug builds expose operator-level counters through RAPID_STATS.

Target Workloads and Use Cases

Use Cases

ShannonBase is designed for organizations that need to consolidate multiple data-platform layers into a single MySQL-compatible system:

Use Cases	ShannonBase capability applied
Real-time operational analytics	HTAP routing: OLTP writes and OLAP reads over live data with millisecond propagation latency
Real-time fraud detection	AutoML classification models served through ML_PREDICT_ROW() with millisecond inference latency
Customer churn prediction	In-database regression/classification training over transaction history, with no data export
Sales forecasting	AutoML time-series forecasting with automated seasonality and trend decomposition
Semantic document search	ML_EMBED() + VECTOR similarity search — a complete in-database RAG pipeline
Recommendation systems	AutoML recommendation (collaborative filtering) + VECTOR nearest-neighbour search
Anomaly detection	Unsupervised anomaly detection over unlabeled operational data (GkNN algorithm)
Feature engineering pipelines	JavaScript MLE stored procedures for preprocessing, transformation and post-inference processing
Generative AI with data sovereignty	In-database LLM inference through ML_GENERATE() — sensitive data never leaves the database

Copyright © 2026, ShannonBase Project Contributors / Public

Conclusion

Compared with conventional MySQL deployments and ordinary HTAP designs, ShannonBase represents a fundamental architectural advance. By embedding its in-house in-memory column engine (Rapid), the self-managing memory automation layer (Autopilot), a fully automated machine-learning pipeline (AutoML), native vector computation, in-database LLM inference and a JavaScript multilingual execution environment directly into the MySQL kernel, ShannonBase removes the architectural fragmentation that forces organizations to operate and synchronize multiple specialized data services.

The system delivers real-time analytical results over live transactional data with millisecond propagation latency; in internal TPC-H reference tests, scan- and aggregation-heavy queries achieve roughly one to two orders of magnitude of speedup over the InnoDB row-store path (the exact factor depends on the query, data volume and concurrency); and it provides enterprise-grade machine learning and generative AI capability inside the database security boundary — with no data export and no dependency on external AP or ML platforms.

ShannonBase is open source at https://github.com/Shannon-Data/ShannonBase. It is built for database kernel developers, data engineers and AI platform architects who need a unified, high-performance, MySQL-compatible foundation for the AI era.

Resources

GitHub repository: https://github.com/Shannon-Data/ShannonBase

License: GPLv2 (MySQL-compatible open-source license)

Supported platforms: Linux x86-64 and aarch64 (Ubuntu 22.04 / 24.04 / 26.04); compatible with the MySQL 8.4 protocol and clients.

Hardware requirements: at least 8 CPU cores and 32 GB of memory for HTAP workloads; 64 GB or more of memory plus a GPU is recommended for production AutoML and Rapid engine deployments.

Copyright © 2026, ShannonBase Project Contributors / Public

Contact Us

Shannon.data.tech@gmail.com, or visit www.shannonbase.org to find the office for your region.

X: https://x.com/ShannonDataAI

Copyright © 2026, ShannonBase Project Contributors / Public