Introducing StrictDB

One API. Every Database. AI-First.

FEBRUARY 26, 2026

THEDECIPHERIST.COM

TABLE OF CONTENTS

I love MongoDB. I've used it in production for years. Native driver, aggregation framework, no Mongoose, no ORM. My containers run at 26 MB. I never think about database problems because there aren't any.

But not every project gets to use MongoDB.

Client wants PostgreSQL. Legacy system runs MySQL. Enterprise mandates MSSQL. Search layer is Elasticsearch. SQLite for local dev. You don't always get to choose. And every time I had to work with a SQL database, I had to context-switch into a completely different mental model. Different syntax. Different driver API. Different error messages. Different everything.

That was annoying. But manageable.

Then AI happened.

THE REAL PROBLEM: AI WRITES TERRIBLE DATABASE CODE

Every AI coding tool — Claude, GPT, Copilot, Cursor, all of them — writes database queries the same way. Inline. Raw. No abstraction. No guardrails. No best practices.

Ask an AI to "add a feature that deletes inactive users" and you'll get something like this:

```
import pg from 'pg';
const pool = new Pool({ connectionString: process.env.DATABASE_URL });
await pool.query('DELETE FROM users WHERE status = $1', ['inactive']);
```

That's a raw SQL string scattered directly in your business logic. No error handling beyond whatever `pg` throws natively. No confirmation that you're about to wipe thousands of rows. No structured receipt telling you what happened. No protection against the AI deciding that `WHERE 1=1` is a valid filter. No logging. Nothing.

And this happens on *every single query the AI writes.* Across every file. Across every feature. You end up with 200 files importing `pg` directly, each one constructing SQL strings with different patterns, different error handling (or none), and different assumptions about the database.

It's the same anti-pattern I spent years fighting against in human-written code. Except now an AI generates it 100x faster and developers accept it without review because "the AI wrote it."

I watched this happen in project after project. AI assistants generating raw `pool.query()` calls, raw `MongoClient` imports, raw `mysql2.execute()` calls — all inline, all scattered, all with zero safety. The AI doesn't know your schema. It guesses column names. It hallucinates table structures. It forgets `LIMIT` clauses. It writes `DELETE FROM` without `WHERE`. And when it throws an error, the AI reads a generic stack trace, guesses at a fix, and usually makes it worse.

This is the problem I built StrictDB to solve.

WHAT STRICTDB ACTUALLY IS

StrictDB is a unified database driver. One API for MongoDB, PostgreSQL, MySQL, MSSQL, SQLite, and Elasticsearch. You write queries in MongoDB's syntax — the same `$gt`, `$in`, `$regex`, `$and`, `$or` operators you already know — and StrictDB translates them to whatever the backend needs. SQL `WHERE` clauses, Elasticsearch Query DSL, native MongoDB operations. The translation is automatic.

```
import { StrictDB } from 'strictdb';

const db = await StrictDB.create({ uri: process.env.DATABASE_URL });

const admins = await db.queryMany('users', {
  role: 'admin',
  status: { $in: ['active', 'pending'] },
  age: { $gte: 18 }
}, { sort: { createdAt: -1 }, limit: 50 });
```

That code runs identically on MongoDB, PostgreSQL, MySQL, MSSQL, SQLite, and Elasticsearch. Change the URI, the code stays the same. MongoDB's query syntax becomes the universal language.

I chose MongoDB's syntax as the foundation because it's already JSON. Your application thinks in JavaScript objects. Your API sends and receives JSON. MongoDB's query operators are native JavaScript — no embedded SQL strings, no template literals, no string concatenation, no parameterized query footguns. Filters are objects. Updates are objects. Everything is objects. It's the most natural fit for how modern applications already work.

And when you're actually running on MongoDB, there's zero translation overhead. The filter passes through as-is. You get native MongoDB performance with the same guardrails and structure that protect you on every other backend.

WHY THIS CHANGES EVERYTHING FOR AI

Here's the part that matters most. StrictDB was built **AI-first** — every design decision assumes an AI agent is calling the API, not just a human.

The AI doesn't need to know what database you're running. It doesn't need to write SQL. It doesn't need to know PostgreSQL uses `$1` parameters while MySQL uses `?` and MSSQL uses `@p1`. It doesn't need to know that MSSQL pagination uses `OFFSET FETCH` instead of `LIMIT`. It doesn't need to remember any of that. It uses one syntax. StrictDB handles the rest.

But the real power is the AI-specific tooling that no other database driver has.

Schema discovery — the AI explores before it queries.

```
const schema = await db.describe('users');
// → { fields: [{ name: 'email', type: 'string', required: true }, ...],
//     indexes, documentCount, exampleFilter }
```

The AI calls `describe()` on a collection and gets back the field names, types, which fields are required, what enums exist, the indexes, the document count, and even an example filter it can use as a starting point. No guessing. No hallucinating column names. No writing `WHERE username = $1` when the column is actually called `email`. The AI knows the schema before it writes a single query.

Dry-run validation — catch the mistake before it hits the database.

```
const check = await db.validate('users', {
  filter: { role: 'admin' },
  doc: { email: 'test@test.com' }
});
// → { valid: true, errors: [] }
```

The AI validates its query before executing it. Wrong field name? Schema mismatch? Invalid operator? It gets caught here. Not in production. Not after the query already ran and corrupted something. Before.

Self-correcting errors — every error tells the AI exactly what to do.

```
catch (err) {
  console.log(err.code);  // 'DUPLICATE_KEY'
  console.log(err.fix);   // 'Use db.updateOne() instead, or check existence wi
}
```

Every `StrictDBError` includes a `.fix` field with the exact corrective action. Not a generic stack trace. Not "something went wrong." A specific, actionable instruction that the AI reads, understands, and acts on. The AI tried to insert a duplicate? The error says "use `updateOne()` instead." The AI used the wrong method? The error says "use `db.queryMany()` instead of `find()`." The collection doesn't exist? The error fuzzy-matches the name and says "did you mean 'users'?"

This is the self-correcting loop. The AI calls the API, gets an error, reads the `.fix` field, adjusts, and succeeds on the next attempt. No human intervention. No stack trace parsing. No ambiguity.

Explain — full transparency on what actually runs.

```
const plan = await db.explain('users', { filter: { role: 'admin' }, limit: 50 }
// → { backend: 'sql', native: 'SELECT * FROM "users" WHERE "role" = $1 LIMIT 5
```

The AI can inspect the translated query before executing it. You (or the AI) can see exactly what SQL gets generated, what Elasticsearch Query DSL gets built, what MongoDB pipeline gets constructed. Nothing hidden. Full transparency.

GUARDRAILS THAT ACTUALLY PROTECT YOUR DATA

This is the "Strict" in StrictDB. These are on by default.

`deleteMany({})` is blocked. Period. An AI that generates an empty filter on a delete operation gets stopped immediately. Not after it wipes your table. Before. The error tells it exactly why it was blocked and what to do instead.

`queryMany` without a `limit` is blocked. No unbounded queries. No "SELECT * FROM users" returning 10 million rows because the AI forgot a LIMIT clause. StrictDB forces a limit on every multi-document read.

`updateMany({})` requires an explicit confirmation flag. If you genuinely want to update every document in a collection, you pass `{ confirm: 'UPDATE_ALL' }`. The AI has to be deliberate about it. No accidents.

Input sanitization runs on every query. SQL column whitelisting. Elasticsearch internal field blocking. Regex complexity limits. Field validation. The pipeline catches bad input before it reaches the database.

I built these because I watched AI tools generate dangerous queries over and over. An AI doesn't get tired. It doesn't hesitate before running a mass delete. It doesn't double-check its WHERE clause. So the guardrails do it for the AI. And for the human developer at 2 AM who's not thinking straight either.

MCP SERVER — 14 TOOLS FOR ANY AI AGENT

StrictDB ships with a Model Context Protocol (MCP) server as a separate package. Connect it to Claude, ChatGPT, or any MCP-compatible agent and it gets 14 self-documenting database tools:

```
strictdb_describe    strictdb_validate    strictdb_explain
strictdb_query_one   strictdb_query_many  strictdb_count
strictdb_insert_one  strictdb_insert_many
strictdb_update_one  strictdb_update_many
strictdb_delete_one  strictdb_delete_many
strictdb_batch       strictdb_status
```

Set `STRICTDB_URI`, start the MCP server, and your AI agent has safe, guardrailed access to any of the 6 supported databases. The AI discovers schemas before querying. Validates before executing. Gets self-correcting errors when something fails. All through one consistent interface regardless of what's running underneath.

This is what "AI-first database access" actually looks like. Not raw SQL strings in a tool call. Not `pool.query()` with fingers crossed. A structured, validated, guardrailed pipeline that the AI can navigate safely.

WHAT STRICTDB IS NOT

It's not an ORM. No models. No migrations. No entity decorators. No lazy loading. No change detection. None of that overhead.

It's not a query builder. You don't chain `.where().orderBy().limit()`. You pass a filter object and options. Done.

It's not Mongoose. It's not Prisma. It's not Drizzle. Those are abstraction layers that add weight, complexity, and overhead between your application and your database. StrictDB is a thin, fast, unified driver. It talks directly to the native database drivers — `mongodb`, `pg`, `mysql2`, `mssql`, `better-sqlite3`, `@elastic/elasticsearch` — and adds exactly three things: translation, guardrails, and structured output. Nothing else.

And when you need to drop down to the raw driver, `db.raw()` is right there. No lock-in.

EVERYTHING ELSE

Structured receipts. Every write operation returns an `OperationReceipt` — `insertedCount`, `modifiedCount`, `deletedCount`, `duration`, `backend`. Never `void`. The AI (and you) always know exactly what happened.

Zod schema validation. Register your schemas with Zod. Enable `schema: true`. Every write gets validated before it touches the database. StrictDB generates `CREATE TABLE` DDL and Elasticsearch mappings from your Zod schemas. One schema definition, every backend.

Batch operations. Mix inserts, updates, and deletes in a single `db.batch()` call. One round trip.

Transactions. `db.withTransaction()` works across MongoDB (replica set), PostgreSQL, MySQL, MSSQL, and SQLite.

Auto timestamps. Configurable `createdAt` / `updatedAt` injection on writes.

Typed events. `connected`, `disconnected`, `reconnecting`, `error`, `operation`, `slow-query`, `guardrail-blocked`, `shutdown`. Hook into whatever you need.

Auto-reconnect. Exponential backoff with configurable attempts, delays, and backoff multiplier.

Zero config. Pass a URI. StrictDB detects the backend automatically. Install only the driver(s) you use — all peer dependencies are optional.

QUICK START

```
npm install strictdb

# Install whatever driver(s) you need
npm install mongodb    # MongoDB
npm install pg         # PostgreSQL
npm install mysql2     # MySQL
```

```
import { StrictDB } from 'strictdb';

const db = await StrictDB.create({ uri: process.env.DATABASE_URL });

// Discover
const schema = await db.describe('users');

// Read
const user = await db.queryOne('users', { email: 'tim@example.com' });

// Write
const receipt = await db.insertOne('users', { email: 'new@example.com', name: '

// Update
await db.updateOne('users', { email: 'tim@example.com' }, { $set: { role: 'admi

// Delete
await db.deleteOne('users', { email: 'old@example.com' });

// What runs under the hood
const plan = await db.explain('users', { filter: { role: 'admin' }, limit: 50 }

await db.close();
```

220 tests passing. MIT licensed. TypeScript-first.

StrictDB.com

npm:

MCP server:

GitHub: