MCP Dev Summit Bengaluru

Why Your Database MCP Should Never Talk to Your Database

A two-layer remote MCP architecture for production database access

Gowtham Raj | Comcast | MCP Dev Summit 2026

Why Not Connect MCP Directly to the DB?

Connecting MCP servers straight to a database works technically — but it gives the LLM (or anyone with MCP access) a direct line to your data layer.

No Per-Agent Access Control	All agents share the same DB credentials. No way to scope which tables or rows each agent can access.
No Query Validation	Raw SQL from an LLM goes straight to your database. Injection is one hallucination away.
No Rate Limiting	No throttling on expensive queries. A runaway scan can take down production.
No Audit Trail	No single chokepoint logging who asked what, when, and how long it took.

It works on a laptop. It fails in production.

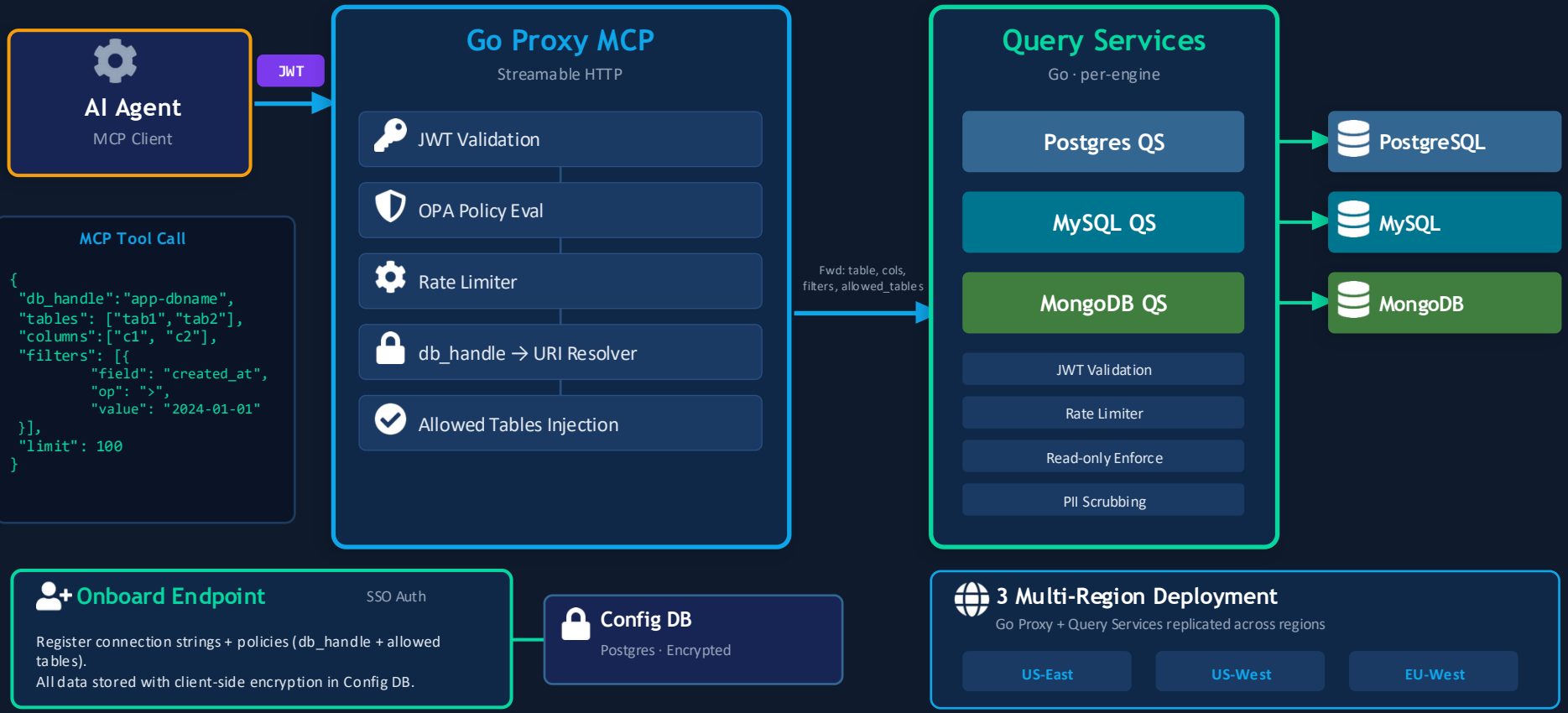
Why a Query API in Between Is Smarter

Put a thin API layer between MCP and your database. MCP calls the API. The API runs the queries.

Access Control	API enforces who can query what — API keys, OAuth, row-level security. MCP only gets the permissions you explicitly grant.
Query Validation	Parameterized queries, allowlists, read-only views. No raw SQL from an LLM hitting your DB. Massively reduces injection risk.
Rate Limiting & Cost	Throttle expensive queries, enforce timeouts, prevent runaway scans that impact production.
Audit Logging	Every query goes through a single chokepoint — who asked what, when, and how long it took.
Abstraction	Swap databases, change schemas, split services — only the API layer changes. MCP server doesn't need to know.
Reusability	Same query API serves MCP, internal dashboards, CLI tools, and other consumers. Build once, use everywhere.

System Architecture

Multi-region · Go Proxy MCP · Query Services · OPA Policy Engine



Sample Payloads: End-to-End Flow

1 Register (Onboard Endpoint)

```
POST /onboard/register
Auth: Bearer <SSO_TOKEN>

{
  "db_handle": "myapp-usersdb",
  "policies": [{
    "name": "analyst-read",
    "allowed_tables": ["users","orders"],
    "allowed_columns": ["id","email"],

  }],
 "postgres_conn_string": "postgres://user:pwd@host/mydb"
}
```

Stored encrypted in Config DB. Teams receive db_handle + policy name to use.

2 Agent → Go Proxy (tools/call)

```
{
  "jsonrpc": "2.0",
  "id": "1",
  "method": "tools/call",
  "params": {
    "name": "execute_sql",
    "arguments": {
      "db_handle": "myapp-usersdb",
      "policy": "analyst-read",
      "table": "users",
      "columns": ["id", "email"],
      "filters": [
        {
          "field": "created_at",
          "op": ">",
          "value": "2024-01-01"
        }
      ],
      "limit": 100
    }
  }
}
```

Proxy validates JWT, evaluates policy via OPA, resolves db_handle, injects allowed_tables.

3 Proxy → Query Service

```
POST /query/execute
Auth: Bearer <SVC_JWT>

{
"db_handle": "myapp-usersdb",
 "table": "users",
 "columns": ["id", "email"],
 "filters": [
  {
   "field": "created_at",
   "op": ">",
   "value": "2024-01-01"
  }
 ],
 "limit": 100
}
```

QS checks table in allowed list, builds parameterized SELECT, scrubs PII.

Generated SQL (Query Service constructs internally — agent never sees this)

```
SELECT id, email FROM users WHERE created_at > $1 LIMIT 100
-- $1 = '2024-01-01'  (parameterized, never concatenated)
```

Query Service: Postgres — Internal Flow

1 Receive Request from Go Proxy

Accepts: db_handle, table, columns[], filters[], limit, allowed_tables[] (injected by proxy via OPA headers)

2 JWT Validation

Validates service-to-service JWT. Confirms request originated from an authenticated Go Proxy instance.

3 Rate Limit Check

Per-db_handle, per-table throttling. Rejects if quota exceeded. Independent from proxy-level rate limit.

4 Table Authorization

Checks requested table against allowed_tables[]. If table not in list → 403 Forbidden. No exceptions.

5 Query Construction (Read-Only)

Builds parameterized SELECT from table, columns, filters, limit. No raw SQL accepted. DDL structurally impossible.

6 Execute on Postgres

Resolves db_handle → connection string. Uses managed pool (read-only user). Statement timeout enforced.

7 PII Scrubbing (Presidio)

Result set scanned for SSNs, emails, phones, names. Redacted to <PERSON>, <EMAIL>, etc. before response.

8 Return Sanitized Results

Clean result set returned to Go Proxy → Agent. No PII, no credentials, no raw connection info in context.

Target: PostgreSQL

Direct Connect vs Two-Layer Architecture

Concern	Direct Connect	Two-Layer MCP
Authentication	None / Basic	JWT + SSO
Credentials in context	Yes — raw URIs	No — execution_id only
DDL protection	None	Blocked at parse level
PII handling	Unfiltered	Presidio scrubbing
Connection mgmt	Per-agent pools	Centralized pooling
Query timeouts	None by default	Enforced structurally
Audit trail	None / ad-hoc	Full execution log
Rate limiting	None	Per-agent throttling

Industry Context: Why Now

OWASP MCP Top 10

Over-privileged access identified as a foundational MCP risk. Agents carrying DB admin credentials is the #1 pattern.

Gateway Architecture Emerging

Microsoft, Docker, AWS all shipping MCP gateways. The industry consensus: enforcement layers between agents and resources.

"Lethal Trifecta" Identified

Agents with private data access + untrusted content exposure + external comms = data exfiltration conduit.

Stdio → Remote Shift

MCP Transport WG moving to Streamable HTTP. Stdio cannot support auth, multi-tenancy, or horizontal scaling.

PII Scrubbing with Microsoft Presidio

BEFORE SCRUBBING

```
{
  "name": "John Smith",
  "ssn": "123-45-6789",
  "email": "john@corp.com",
  "phone": "555-867-5309",
  "balance": "$45,230"
}
```

AFTER SCRUBBING

```
{
  "name": "<PERSON>",
  "ssn": "<US_SSN>",
  "email": "<EMAIL>",
  "phone": "<PHONE>",
  "balance": "$45,230"
}
```

Presidio uses NLP + regex + context-aware detection to identify PII entities across 50+ categories. Non-PII fields (like balance) pass through unchanged — agents still get the data they need for analysis.

Scrubbing happens in the Query Service — before results enter any agent context window. The LLM never sees raw PII.

Threat Mitigation Summary

Threat	How We Mitigate
Credential Leakage	execution_id replaces raw URIs. Proxy holds credentials — agents never see them.
SQL Injection	Query Service parses and parameterizes all SQL before execution.
DDL / Schema Mutation	Statement-level blocklist at parse time. Not role-based — structural.
PII Exposure to LLM	Presidio scrubs result sets. 50+ entity types detected and redacted.
Connection Exhaustion	Centralized pooling in Query Service. Rate limiting in Proxy.
Runaway Queries	Statement timeouts + max row limits enforced at Query Service level.
Prompt Injection → DB	Two layers of indirection. Agent can't construct raw DB commands.
No Audit Trail	Every query logged with execution_id, user identity, timestamp, results.

Key Takeaways

An enforcement layer between MCP and your database is the architecture — not an option.

db_handle is a reusable enterprise primitive. Any resource, any agent, zero credential exposure.

Separation of concerns: Proxy handles identity. Query Service handles data. Neither does both.

Defense in depth: every threat addressed structurally, not through policy or hope.

Thank You

Questions & Discussion

The enforcement layer isn't optional.

It's the architecture.

MCP Dev Summit Bengaluru