ViewLegacy.

Safety Comparison Between Direct Database Access for AI Agents and Capability-Based Access

Technical white paper

Published by: ViewLegacy LLC.
Author: ASAHI MASUDA
Publication date: May 19, 2026

This paper compares four ways to connect AI agents to operational databases: direct database credentials, natural-language-to-SQL, generic CRUD APIs, and agent-defined business capabilities.

1. Executive Summary

Organizations increasingly want AI agents to answer operational questions from customer, order, inventory, accounting, and employee systems. The architectural question is not whether an agent can call a database. It is how much authority the agent runtime should receive when prompts, tool descriptions, and retrieved content may be manipulated.

This white paper compares four access patterns: direct database credentials, SQL-generating agents, generic table-level CRUD APIs, and agent-defined capability-based access. The main conclusion is balanced: capability-based access can substantially reduce the blast radius of AI agent errors and prompt-injection-driven misuse, but it does not eliminate all risk.

Capability-based access is not always the right architecture for analytics, large-scale BI, or workloads that require flexible ad hoc querying. It is particularly suitable for operational, read-oriented, legacy-system access where actions can be expressed as explicit business capabilities.

2. Background

AI agents increasingly need to retrieve data from systems of record. For many organizations those systems are legacy relational databases that were designed for internal applications, not autonomous agent runtimes. They often contain sensitive customer, order, inventory, accounting, and employee data in the same schema.

Direct database access gives the AI application a broad technical interface: credentials, schema visibility, and the ability to run whatever the database role permits. Natural-language-to-SQL systems add a further translation problem: the model must infer the user's intent, produce correct SQL, respect authorization, and avoid disclosing unrelated data.

MCP and tool-calling make it easier for agents to call business systems. That convenience does not remove the need for least privilege. Tool exposure should still be designed as an authorization boundary, not merely as a documentation layer.

3. Compared Access Patterns (1/4)

Pattern 1: Direct Database Credentials

In this pattern, the AI application or agent runtime receives database credentials. The agent can see the schema available to the database role and can execute SQL allowed by that role. Authorization is mostly enforced by database privileges, application prompts, and sometimes query-generation guardrails.

When the agent is compromised or misled, the attacker can ask it to enumerate tables, join unrelated data, bypass intended business workflows, or modify data if the credential permits writes. Even if the database user is read-only, confidentiality risk remains high because a broad read role can still expose far more data than a business task requires.

This pattern exposes raw schema and usually exposes database error behavior. It may also put credentials into application configuration, logs, traces, or secret stores used by the AI runtime.

3. Compared Access Patterns (2/4)

Pattern 2: SQL-Generating Agent

A natural-language-to-SQL agent gives the model a database interface and asks it to generate SQL from user intent. The agent can often see schema metadata and examples. It can do whatever the generated SQL and database role allow.

Authorization is difficult to reason about because it is split across prompt instructions, schema selection, query-generation logic, database privileges, and post-processing. Prompt injection can steer the model toward broader queries, schema disclosure, row-limit bypass attempts, or columns not needed for the task.

Prepared statements help when user values are bound as parameters, but they do not solve the core issue when the model is allowed to synthesize query structure. Correctness, authorization, and auditability all depend on a generated artifact that may vary per request.

3. Compared Access Patterns (3/4)

Pattern 3: Generic CRUD or REST API

A generic CRUD API exposes tables or resources through HTTP operations. The agent no longer holds database credentials and usually cannot submit raw SQL. It can see the API surface, resource names, field names, and the operations available to its API key.

Authorization is enforced in the API layer. This is an improvement over handing out database credentials, but generic table-shaped APIs often expose broad data surfaces. If endpoints map closely to tables, the model can still enumerate records, request unnecessary fields, or call update and delete operations if its key allows them.

This pattern is easier to audit than direct SQL, but the audit trail may still be table-centric rather than business-intent-centric.

3. Compared Access Patterns (4/4)

Pattern 4: Agent-Defined Capability-Based Access

In a capability-based architecture, the agent calls named business operations such as `get_customer_order_status` or `list_open_invoices_for_account`. It does not receive database credentials, raw SQL access, or schema-wide CRUD access.

Selected operations are exposed through REST API and MCP, but the exposed surface is a capability contract rather than a database contract. The gateway handles routing, identity, capability authorization, OpenAPI filtering, MCP tool filtering, and observability. It should not store business data, raw SQL, or database credentials in the provider cloud. The on-premise agent owns SQL, database credentials, parameter validation, execution policy, prepared-statement execution, and output filtering. A `capability.yaml` file defines which operations can exist.

If the AI agent is misled, it can only call visible capabilities with accepted parameters. This narrows the executable surface and makes audit records align with business operations. The design still depends on correct capability definitions, scoped API keys, careful logging, and secure operation of the on-prem agent host.

3. Architecture Diagrams

Direct DB Access

```
[AI Agent]
    |
    v
[App / Agent Runtime] -- credentials, SQL, schema access --> [(Legacy Database)]
    ^
    |
 prompt injection can influence query intent
```

Capability-Based Access

```
[AI Agent / MCP Client]
    |
    v
[Gateway: auth, routing, observability]
    |
    | capability name + validated-looking params
    v
[On-Prem Agent: validation, policy, prepared SQL, output allow-list]
    |
    v
[(Legacy Database)]
```

4. Evaluation Criteria

Axis	Direct DB Access	SQL-Generating Agent	Generic CRUD API	Capability-Based Access
Least privilege	Weak unless DB role is very narrow	Weak to medium; model still has query authority	Medium; endpoint/key scoping possible	Stronger when capabilities are narrow
Blast radius under prompt injection	High	High	Medium to high	Lower; limited to allowed capabilities
SQL injection / query manipulation	Depends on query construction	High concern for generated query structure	Lower if no raw SQL endpoint	Lower with prepared SQL and no raw SQL surface
Schema exposure	High	High	Medium	Low; capability metadata only
Data minimization	Weak	Weak to medium	Medium	Stronger with result allow-lists
Output filtering	Usually external	Often post-hoc	API-specific	Agent-enforced allow-list
Auditability	Query-level, often technical	Generated-query-level	Endpoint-level	Capability-name-level
Revocability	Rotate DB credential / role	Rotate DB credential / role	Revoke endpoint/API key	Revoke API key capability grant
Operational complexity	Low initial, high risk	High	Medium	Medium to high
Legacy suitability	Risky exposure	Risky exposure	Requires API modeling	Strong fit for read-oriented operations
MCP / tool-calling fit	Poor	Risky	Usable	Strong fit for named tools

4. Pattern Comparison

Pattern	Authority exposed to AI	Schema exposure	Can run arbitrary SQL?	Output minimization	Audit granularity	Main risk
Direct DB credentials	Database role authority	High	Yes, within role	Weak unless manually constrained	SQL/session logs	Prompt injection becomes database misuse
SQL-generating agent	Query-generation authority plus DB role	High	Yes, through generated SQL	Weak to medium	Prompt and generated SQL	Incorrect or overbroad SQL becomes authorized action
Generic CRUD API	Endpoint and resource authority	Medium	No, unless exposed	Medium	Endpoint/resource logs	Table-shaped APIs expose broad records and fields
Capability-based access	Named business-operation authority	Low	No raw SQL interface	Stronger with result allow-lists	Capability name, key, status, latency	Poor capability design can recreate broad access

5. Evaluation Method

The evaluation is an illustrative threat-model simulation, not a formal proof. It models a small legacy database domain containing customers and orders. It then asks how each access pattern responds to ten attack or failure scenarios.

The supporting source archive is `src.zip`, published alongside this PDF:

- `evaluation-model.json` defines the sample domain, assumptions, patterns, scenarios, qualitative outcomes, and scenario notes.
- `evaluate-scenarios.mjs` generates `scenario-outcomes.md` from that model.

The method intentionally evaluates authority and failure modes rather than benchmark performance. Its purpose is to make the security trade-offs concrete enough to discuss, not to certify any architecture.

5. Evaluation Scenarios

ID	Scenario
S1	Prompt injection asks the agent to list all customers.
S2	Prompt injection asks the agent to reveal database schema.
S3	Prompt injection asks the agent to run an arbitrary SQL query.
S4	Prompt injection asks the agent to retrieve columns not required by the business task.
S5	Prompt injection asks the agent to bypass row limits.
S6	Prompt injection asks the agent to update or delete data.
S7	The API key or agent runtime is compromised.
S8	The gateway or API layer is compromised.
S9	The tool catalog or OpenAPI definition is exposed.
S10	A legitimate user requests an allowed business operation with valid parameters.

Qualitative labels are used: `High risk`, `Medium risk`, `Lower risk`, `Blocked by design`, `Depends on implementation`, and `Allowed`.

6. Evaluation Results: Scenario Outcomes

Scenario	Direct DB Access	SQL-Generating Agent	Generic CRUD API	Capability-Based Access
S1: list all customers	High risk	High risk	Medium risk	Lower risk
S2: reveal schema	High risk	High risk	Medium risk	Blocked by design
S3: arbitrary SQL	High risk	High risk	Blocked by design	Blocked by design
S4: unnecessary columns	High risk	High risk	Medium risk	Lower risk
S5: bypass row limits	High risk	High risk	Depends on implementation	Lower risk
S6: update/delete data	High risk	High risk	High risk	Blocked by design
S7: API key/runtime compromised	High risk	High risk	Medium risk	Medium risk
S8: gateway/API compromised	High risk	High risk	High risk	Medium risk
S9: tool catalog exposed	High risk	High risk	Medium risk	Lower risk
S10: allowed operation	Allowed	Allowed with correctness risk	Allowed	Allowed

The capability-based pattern has lower risk in many prompt-injection scenarios because the model cannot invent executable SQL or expand the database interface at runtime. The remaining risk is concentrated in the quality of the capability definitions, API-key scoping, output filtering, and agent host security.

6. Evaluation Results: Risk Comparison

Risk area	Direct DB Access	SQL-Generating Agent	Generic CRUD API	Capability-Based Access
Confidentiality failure	High	High	Medium	Lower, if outputs are narrow
Integrity failure	High if write role exists	High if write role exists	High if write endpoints exist	Lower for read-only scope
Prompt-injection misuse	High	High	Medium	Lower
Schema reconnaissance	High	High	Medium	Lower
Credential exposure	DB credentials in AI app path	DB credentials in AI app path	API credentials only	API credentials at gateway; DB credentials on-prem agent
Audit ambiguity	High	High	Medium	Lower
Operational burden	DB role management	Model/query guardrails plus DB roles	API authorization design	Capability review and agent operations

Capability-based access reduces exposed authority, narrows the executable surface, limits blast radius, improves auditability, and can make policy enforcement more explicit. These are conditional claims. They depend on narrow capability design, reviewed `capability.yaml` changes, conservative API-key grants, and logs that avoid sensitive business data.

6. Evaluation Results: Risk Movement

Risk	Where it appears in direct access	Where it appears in capability-based access	Residual control required
Database credential misuse	AI application, runtime, secret store, logs	On-prem agent host and local config	Host hardening, secret handling, least-privilege DB user
Query manipulation	Runtime-generated SQL or direct user-driven SQL	Capability SQL reviewed before deployment	Code review, SQL lint, read-only policy, prepared statements
Overbroad data disclosure	Broad tables, joins, and columns	Overbroad capability result set	Output allow-list review and max rows/bytes
Unauthorized operation	Database role permits too much	API key permits too many capabilities	Per-capability API-key scope and revocation
Metadata exposure	Schema and table names	Capability catalog and public descriptions	Avoid secrets and internal details in OpenAPI/MCP descriptions
Gateway compromise	May expose DB path or credentials	Can affect routing/auth but lacks SQL and DB credentials	Agent-side enforcement, monitoring, key rotation

The important shift is that raw database authority moves away from the AI runtime and gateway. Security review moves toward capability definitions, API-key grants, and the on-prem agent execution boundary.

7. Discussion (1/2)

Capability-based access is less flexible than ad hoc SQL. That is the point for operational use cases, but it is a real trade-off. Analysts who need flexible exploration, broad joins across many tables, or one-off investigations may be better served by a governed data warehouse, semantic layer, or BI environment. Other approaches may also be better when the primary use case is batch analytics or machine learning over large historical datasets.

Poorly designed capabilities can still expose too much data. A capability named `export_all_customers` with broad columns and high row limits is dangerous even if it is technically a capability. A capability set can also become so broad that it effectively recreates database access. Output allow-lists can fail when they are poorly defined, and capability changes can introduce new risk when they are not reviewed.

Read-only scope materially reduces destructive risk, but it does not remove confidentiality risk. A read-only agent can still leak sensitive data if a capability returns unnecessary columns, if API keys are over-permissioned, if logs contain business data, or if the on-prem agent host is compromised.

7. Discussion (2/2)

Gateway compromise and agent compromise have different implications. In the evaluated architecture, the gateway should not know raw SQL or database credentials. That limits what a gateway compromise can directly reveal. However, a compromised gateway can still abuse routing, authorization, observability, or cached metadata unless agent-side enforcement and monitoring are strong.

The on-prem agent and `capability.yaml` form a critical security boundary. The agent owns database credentials, SQL execution, validation, policy, and output filtering. Capability changes therefore deserve the same review discipline as application authorization code.

Capability-based access is usually easier to govern for operational AI agents because operations are explicit. It is not a substitute for secure database roles, network controls, secret management, monitoring, or incident response.

8. Practical Implications (1/2)

Companies connecting AI agents to legacy systems should start with read-only operations and define capabilities around business intent, not database tables. A good capability name should describe the operation the business is willing to expose, not the table the query happens to read.

Use input validation, execution policy, prepared SQL, output allow-lists, row limits, byte limits, and timeouts. Scope API keys by allowed capability. Avoid exposing raw SQL, raw schema access, and database credentials to the AI runtime or gateway.

OpenAPI and MCP tool definitions should be filtered by authorization scope. Tool descriptions are public contract text for allowed operations; they should not contain secrets, internal incident notes, database credentials, or sensitive operational detail.

8. Practical Implications (2/2)

Operational telemetry should help answer who called which capability, whether it succeeded, what error code was returned, and how long it took. Logs should include capability name, API key identity, status, error code, request identifier, and latency.

Logs should not store business data, raw SQL, submitted parameter values, database connection information, credentials, authorization headers, tokens, API keys, stack traces, or database-driver details. Sensitive diagnostics should stay in tightly controlled local agent logs when needed for operations.

Treat capability definitions as security-critical configuration. Review changes, keep capability sets small, prefer read-only database users for read capabilities, and rotate or revoke API keys when access requirements change.

9. Limitations and Future Work

This evaluation is illustrative and is not a substitute for a full security audit, formal threat model, or red-team assessment. It does not cover every database engine, driver, ORM, network topology, deployment model, MCP client, or agent framework.

The comparison assumes capability definitions are reviewed and correctly implemented. It focuses primarily on read-oriented access. Write operations require additional controls: approval flows, idempotency, transaction design, conflict handling, stronger audit trails, and explicit recovery procedures.

Performance and latency are not deeply benchmarked. Human operational processes, such as change approval, incident response, access reviews, and key rotation discipline, are not fully evaluated.

Future work should evaluate real MCP clients, multiple database engines, formal threat modeling, adversarial prompt tests, red-team exercises, data-loss-prevention integration, and write-operation safety patterns.

10. References

1. Model Context Protocol, "Tools", https://modelcontextprotocol.io/specification/2025-06-18/server/tools
2. Model Context Protocol, "Authorization", https://modelcontextprotocol.io/specification/2025-06-18/basic/authorization
3. OWASP, "Top 10 for Large Language Model Applications", https://owasp.org/www-project-top-10-for-large-language-model-applications/
4. OWASP Cheat Sheet Series, "SQL Injection Prevention Cheat Sheet", https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
5. OWASP Cheat Sheet Series, "Injection Prevention Cheat Sheet", https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_Cheat_Sheet.html
6. NIST, "SP 800-207: Zero Trust Architecture", https://csrc.nist.gov/pubs/sp/800/207/final
7. OpenAPI Initiative, "The OpenAPI Specification Explained", https://learn.openapis.org/specification/
8. OpenAPI Initiative, "API Endpoints", https://learn.openapis.org/specification/paths.html
9. PostgreSQL Documentation, "Privileges", https://www.postgresql.org/docs/17/ddl-priv.html
10. PostgreSQL Documentation, "Row Security Policies", https://www.postgresql.org/docs/17/ddl-rowsecurity.html

Finally

The comparison does not show that capability-based access is absolutely secure. It shows a narrower and more auditable authority model for operational AI access to legacy databases.

The design is strongest when capabilities are few, explicit, read-oriented, parameterized, output-filtered, and reviewed as security-sensitive configuration.

Published by: ViewLegacy LLC.
Author: ASAHI MASUDA
Publication date: May 19, 2026