ViewLegacy.

Least-Privilege Design for Connecting Business Systems in the MCP Era

Technical white paper

Published by: ViewLegacy LLC.
Author: ASAHI MASUDA
Publication date: June 5, 2026

This paper evaluates how operational business systems can be exposed to AI agents through MCP and tool-calling interfaces without handing the agent broad database, schema, CRUD, or credential authority.

1. Executive Summary

MCP makes it easier for AI agents to discover and call external tools. That interoperability is valuable, but it changes the security question from "Can the agent reach the system?" to "Which operations can the agent discover, call, parameterize, and observe?"

This paper compares five exposure patterns: direct SQL tools, generic table-level CRUD tools, broad internal API catalogs, role-filtered API or tool catalogs, and agent-defined capability-based access with filtered MCP tool discovery.

The main conclusion is balanced. Least privilege should apply not only to runtime execution, but also to tool discovery, schema exposure, input parameters, output fields, row limits, logging, and audit visibility. Capability-based access can reduce exposed authority by presenting explicit business operations rather than broad database or table access.

It does not eliminate all risk. It is not always the right architecture for analytics, large-scale BI, or 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 access to operational data: customer status, order history, inventory position, invoice status, support context, and internal workflow state. In many organizations, that data still lives in legacy databases or applications that were not designed for autonomous agent runtimes.

MCP standardizes how clients discover tools and call external systems. In the MCP tools model, a client can request `tools/list` and then invoke a named tool with structured arguments through `tools/call`. Tool definitions include names, descriptions, and schemas. This is useful for interoperability, but the catalog itself can reveal sensitive business capabilities, entity names, workflows, schemas, and operational structure.

Legacy databases often contain sensitive customer, order, inventory, accounting, and employee data in the same environment. Exposing raw SQL access, schema-wide CRUD APIs, or unfiltered OpenAPI and MCP definitions can violate least privilege before any runtime call is made.

Prompt injection and tool misuse make the boundary more important. Least privilege must be enforced outside the model, not merely requested in a prompt.

3. Compared Exposure Patterns (1/5)

Pattern 1: Direct database or SQL tool

The AI agent can discover a raw database or SQL tool such as `run_sql`. It can call that tool with queries constrained primarily by the database credentials, database role, SQL parser, and any wrapper-level restrictions.

The discoverable surface can include table names, schema descriptions, query examples, database error behavior, or introspection capabilities. Authorization is usually enforced in the database role and sometimes in the tool wrapper.

If the agent is compromised, misled, or instructed maliciously, the attacker can attempt broad table enumeration, joins across unrelated business domains, row-limit bypasses, or write operations if the credential permits them. Even a read-only database role can still expose far more data than a specific business task requires.

3. Compared Exposure Patterns (2/5)

Pattern 2: Generic table-level CRUD tools

The AI agent can discover tools shaped around tables or resources, such as `list_customers`, `get_order`, `update_invoice`, or `delete_order`. The agent may not hold database credentials or raw SQL access, but the tool catalog still exposes table-shaped business structure.

The callable surface is broader than a specific business workflow. Authorization is typically enforced in the API layer by endpoint, role, object ownership, or resource policy. Tool definitions may or may not be filtered before discovery.

If the agent is compromised or misled, it may call unrelated table operations, request unnecessary fields, enumerate records, or use write/delete tools that were exposed for a different application context.

3. Compared Exposure Patterns (3/5)

Pattern 3: Broad internal API catalog

The AI agent can discover a large set of internal APIs exposed as MCP tools or OpenAPI-derived functions. This avoids raw database credentials, but it can still reveal a detailed map of internal systems, entity names, workflows, and administrative functions.

The callable surface depends on the API key or user identity. In many deployments, discovery is broader than execution: the agent can see endpoints that later fail with `403 Forbidden`.

This is better than raw SQL when APIs are well designed, but it can overwhelm the model and increase accidental misuse. If an agent is maliciously instructed to search for sensitive workflows, the catalog itself may provide useful reconnaissance even when execution is blocked.

3. Compared Exposure Patterns (4/5)

Pattern 4: Role-filtered API or tool catalog

The AI agent can discover only tools or endpoints associated with the caller's role, user identity, tenant, or API-key scope. Discovery filtering reduces unnecessary visibility, and runtime authorization rejects calls outside the granted scope.

This is a meaningful improvement over broad catalogs. It limits tool descriptions, reduces accidental misuse, and makes `tools/list` part of the authorization boundary.

The design still depends on correct role definitions, exact catalog filtering, and independent execution checks. If discovery filtering and execution authorization drift apart, a hidden tool may become callable or a visible tool may disclose more than intended.

3. Compared Exposure Patterns (5/5)

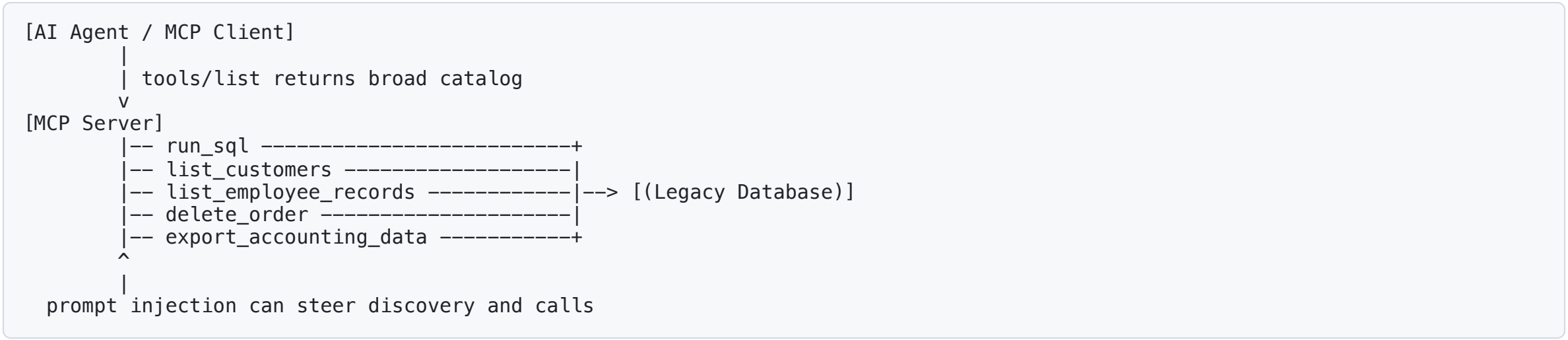
Pattern 5: Agent-defined capability-based access

In this pattern, the AI agent discovers and calls named business capabilities, not raw SQL, raw schema access, broad table CRUD, or database credentials. Examples include `get_customer_summary`, `search_orders`, `check_inventory`, and `get_invoice_status`.

Selected operations are exposed through REST API and MCP. API keys are scoped to allowed capabilities. OpenAPI and MCP tool definitions are filtered based on caller authorization. The cloud gateway handles routing, identity, authorization, and observability, but should not know raw SQL or database credentials.

The on-prem agent owns SQL, database credentials, validation, execution policy, prepared SQL execution, output filtering, row limits, and business meaning. A `capability.yaml` file defines the operations that can exist. In the initial read-only scope, destructive operations are out of scope by design.

3. Diagram: Broad MCP Tool Exposure



In broad exposure, tool discovery may reveal sensitive operations before the caller attempts execution. Blocking `tools/call` is necessary, but it does not remove metadata exposure from an unfiltered catalog.

3. Diagram: Least-Privilege Discovery and Execution

```
[AI Agent / MCP Client]
        |
        | tools/list
        v
[Gateway: identity, auth, tool filtering, observability]
        |
        | filtered tool definitions
        v
[AI Agent / MCP Client]
        |
        | tools/call: capability + params
        v
[Gateway: runtime authorization]
        |
        | authorized capability
        v
[On-Prem Agent: validation, policy, prepared SQL, output allow-list]
        |
        v
[(Legacy Database)]
```

The gateway narrows what the agent can discover and call. The agent-side execution boundary then validates parameters and filters results before data returns to the caller.

3. Diagram: Authorization Scope Example

```
sales_partner key
  |-- get_customer_summary
  +-- search_orders

support_agent key
  |-- get_customer_summary
  |-- search_orders
  |-- get_invoice_status
  +-- check_inventory

internal_admin key
  |-- get_customer_summary
  |-- search_orders
  |-- get_invoice_status
  |-- check_inventory
  +-- list_employee_records

No scoped key exposes run_sql or delete_order in the read-only service scope.
```

This model treats tool descriptions as security-sensitive. A partner key should not learn that employee-record tools exist unless that business exposure is intentional.

3. Exposure Pattern Comparison

Pattern	Discoverable surface	Callable surface	Schema/business-process exposure	Authorization model	Main risk
Direct SQL tool	Raw SQL tool, schema notes, examples, errors	Any SQL permitted by DB role and wrapper	High: tables, columns, joins, errors	Database role plus wrapper checks	Prompt injection becomes database misuse
Generic CRUD tools	Table/resource-shaped tools	Broad table operations by API key	Medium to high: resource and field names	API endpoint and object checks	CRUD surface exceeds business need
Broad API catalog	Many internal APIs and descriptions	Depends on API key or user	High: internal workflows and admin functions	API gateway and service auth	Catalog leaks operational structure
Role-filtered catalog	Role-scoped tools/endpoints	Role-scoped operations	Medium: allowed business tools only	Discovery filter plus runtime auth	Scope drift or overbroad roles
Capability-based MCP access	Caller-scoped business capabilities	Explicit capability calls only	Lower: business operations, not schema	API-key scope, filtered discovery, runtime auth, agent policy	Poor capability design can recreate broad access

4. Evaluation Criteria: Discovery and Runtime

Axis	Direct SQL Tool	Generic CRUD Tools	Broad API Catalog	Role-Filtered Catalog	Capability-Based MCP Access
Tool discovery minimization	Weak	Weak to medium	Weak	Stronger	Stronger
Runtime least privilege	Weak unless DB role is very narrow	Medium if endpoints are narrow	Medium, varies by API	Stronger	Stronger when capabilities are narrow
Schema and business-process exposure	High	Medium to high	High	Medium	Lower
Parameter-level constraints	Weak to medium	Medium	Medium	Medium	Stronger: capability-specific validation
Output allow-listing	Usually external or post-hoc	Endpoint-specific	Service-specific	Service-specific	Agent-enforced allow-list
Row, byte, and timeout limits	Depends on wrapper and DB	Depends on API	Depends on API	Depends on API	Execution-policy boundary
Authorization by API key or identity	DB role or wrapper	API key/user	API key/user	Role/key scope	Capability-scoped key plus runtime check

4. Evaluation Criteria: Security and Operations

Axis	Direct SQL Tool	Generic CRUD Tools	Broad API Catalog	Role-Filtered Catalog	Capability-Based MCP Access
Prompt-injection blast radius	High	Medium to high	Medium to high	Lower	Lower when correctly implemented
Auditability by business operation	Weak: query/session logs	Medium: table/resource logs	Medium: endpoint logs	Stronger	Stronger: capability name, caller, status
Revocation and key rotation	DB credential or wrapper key	API key	API key	Scoped API key	Capability-scoped API key
Operational complexity	Low start, high risk	Medium	Medium to high	Medium to high	Medium to high
Suitability for legacy systems	Risky exposure	Requires API modeling	Often too broad	Good if well scoped	Strong for read-oriented operations
Suitability for MCP/tool-calling agents	Poor	Usable but broad	Usable but noisy	Better	Strong fit for explicit named tools
Failure mode under compromised agent	Database authority exposed	Broad resource calls	Broad API reconnaissance	Scoped misuse	Scoped misuse plus residual data risk

5. Evaluation Method

The evaluation is an illustrative threat-model simulation, not a formal proof. It does not implement a full MCP server and does not connect to a production database.

The model is stored under `src.zip`:

- `evaluation-model.json` defines a fictional business system, tools, roles, exposure patterns, scenarios, and qualitative outcomes.
- `evaluate-scenarios.mjs` generates `scenario-outcomes.md` from that model.
- `scenario-outcomes.md` records role-based discovery and scenario outcomes used in this paper.

The fictional tools include `get_customer_summary`, `search_orders`, `check_inventory`, `get_invoice_status`, `list_employee_records`, `run_sql`, and `delete_order`. The roles are `sales_partner`, `support_agent`, `internal_admin`, and an intentionally unsafe `over_permissioned_key`.

The method evaluates authority and blast radius, not performance or formal exploitability.

5. Evaluation Scenarios

ID	Scenario
S1	A sales partner calls `tools/list`.
S2	A support agent calls `tools/list`.
S3	A compromised agent attempts to discover admin-only tools.
S4	A prompt injection asks the agent to call an unrelated business tool.
S5	A prompt injection asks the agent to reveal all available tools.
S6	A prompt injection asks the agent to call a raw SQL tool.
S7	A legitimate user calls an allowed business capability with valid parameters.
S8	A user attempts to request more rows than allowed.
S9	A user attempts to request fields outside the output allow-list.
S10	An API key is over-permissioned with access to all tools.

Qualitative labels are used: `High risk`, `Medium risk`, `Lower risk`, `Blocked by design`, `Blocked if correctly implemented`, `Depends on implementation`, `Allowed`, and `Out of scope`.

6. Evaluation Results: Tool Discovery by Role

Role / API key	Expected tools	Should not discover	Risk if unfiltered
`sales_partner`	`get_customer_summary`, `search_orders`	`check_inventory`, `get_invoice_status`, `list_employee_records`, `run_sql`, `delete_order`	Partner learns support, HR, raw SQL, or write capabilities exist.
`support_agent`	`get_customer_summary`, `search_orders`, `get_invoice_status`, `check_inventory`	`list_employee_records`, `run_sql`, `delete_order`	Support context exposes admin, raw SQL, or destructive tools.
`internal_admin`	`get_customer_summary`, `search_orders`, `get_invoice_status`, `check_inventory`, `list_employee_records`	`run_sql`, `delete_order`	Admin scope drifts into raw database or destructive authority.
`over_permissioned_key`	All tools	None	Least privilege has effectively failed; any compromised agent can discover and call the full catalog.

Discovery filtering matters because tool names and descriptions can disclose business structure even when execution is later denied.

6. Evaluation Results: Scenario Outcomes

Scenario	Direct SQL Tool	Generic CRUD Tools	Broad API Catalog	Role-Filtered Catalog	Capability-Based MCP Access
S1: sales partner calls `tools/list`	High risk	High risk	High risk	Lower risk	Lower risk
S2: support agent calls `tools/list`	High risk	High risk	High risk	Lower risk	Lower risk
S3: compromised agent discovers admin-only tools	High risk	High risk	High risk	Blocked if correctly implemented	Blocked if correctly implemented
S4: injection calls unrelated tool	High risk	Medium risk	Medium risk	Blocked if correctly implemented	Blocked if correctly implemented
S5: injection reveals all tools	High risk	High risk	High risk	Lower risk	Lower risk
S6: injection calls raw SQL	High risk	Blocked by design	Depends on implementation	Blocked if correctly implemented	Blocked by design
S7: valid allowed capability	Allowed	Allowed	Allowed	Allowed	Allowed
S8: request too many rows	High risk	Depends on implementation	Depends on implementation	Depends on implementation	Blocked if correctly implemented
S9: request unapproved fields	High risk	Medium risk	Medium risk	Depends on implementation	Blocked if correctly implemented
S10: over-permissioned key	High risk	High risk	High risk	High risk	High risk

6. Evaluation Results: Risk Comparison

Risk area	Direct SQL Tool	Generic CRUD Tools	Broad API Catalog	Role-Filtered Catalog	Capability-Based MCP Access
Discoverable authority	High	High	High	Lower	Lower
Callable authority	High	Medium to high	Medium to high	Lower if scoped	Lower if capabilities are narrow
Confidentiality failure	High	Medium to high	Medium to high	Medium	Lower, not eliminated
Integrity failure	High if writes allowed	High if write tools exposed	High if write APIs exposed	Lower if writes filtered	Lower in read-only scope
Prompt-injection blast radius	High	Medium to high	Medium to high	Lower	Lower
Schema reconnaissance	High	Medium	Medium to high	Lower	Lower
Audit ambiguity	High	Medium	Medium	Lower	Lower
Key over-permissioning impact	High	High	High	High	High

Least-privilege MCP design must filter both discovery and execution. Filtering only `tools/call` leaves catalog metadata exposed. Filtering only `tools/list` is insufficient if hidden tools remain callable by name.

6. Evaluation Results: Risk Movement

Risk	Where it appears in broad MCP exposure	Where it appears in least-privilege capability-based MCP	Residual control required
Raw database authority	AI runtime, SQL tool, schema prompts	On-prem agent and local DB role	Host hardening, secret handling, DB least privilege
Tool reconnaissance	Full catalog and descriptions	Caller-filtered capability catalog	Review descriptions as security-sensitive content
Unauthorized operation	Agent sees or calls unrelated tools	Gateway and agent authorization boundary	Filter discovery and enforce runtime authorization
Overbroad output	Tables, fields, broad API responses	Capability output allow-list	Review fields, row limits, byte limits
Prompt injection	Model can steer broad calls	Model can steer only granted capabilities	External enforcement, monitoring, human review where needed
Bad configuration	Broad API key exposes many APIs	Over-permissioned key exposes many capabilities	Key scoping, rotation, access review
Sensitive logging	Prompts, SQL, params, API results	Gateway and agent logs	Avoid raw data, SQL, credentials, tokens, and stack traces

The risk does not disappear. It moves from the MCP client/model layer toward gateway authorization, capability definition, agent execution policy, and operations.

6. Evaluation Results: Interpretation

The evaluation supports a limited claim: capability-based MCP access can reduce exposed authority when discovery filtering, runtime authorization, parameter validation, output allow-listing, row limits, and logging discipline are correctly implemented.

It does not show that the architecture is secure in an absolute sense. A broad capability can still leak data. A single overpowered API key can undo careful tool design. A compromised on-prem agent host can expose the database path and local credentials. Poor logs can leak sensitive business data.

The main advantage is governance clarity. Instead of asking whether an agent should "access the database," reviewers can ask whether a caller should discover and call a specific business capability, with specific parameters, specific output fields, and explicit limits.

7. Discussion (1/2)

Filtering `tools/list` is as important as filtering `tools/call`. Tool descriptions can leak business structure even if execution is blocked. A catalog that exposes `list_employee_records`, `refund_invoice`, or `delete_order` tells the model and the user that those workflows exist.

Broad API catalogs can also overwhelm AI agents. The more unrelated tools the model can see, the more opportunities it has to choose the wrong tool or follow a malicious instruction into an unintended workflow.

Capability definitions become a critical security boundary. A capability should be narrow enough to represent a business operation, not so broad that it recreates database access. A capability named `export_all_customers` with many fields and high row limits is not meaningfully least privilege.

Read-only scope materially reduces destructive risk. It does not remove confidentiality risk.

7. Discussion (2/2)

MCP improves interoperability, but interoperability does not automatically imply safe exposure. The protocol can describe tools and calls; it does not decide which tools a business should publish to which caller.

For analytical workloads, governed data warehouses, BI semantic layers, or controlled analyst environments may be more appropriate. That is especially true when analysts need flexible ad hoc exploration, broad joins across many tables, batch analytics, machine learning over large historical datasets, or established human-governed exploratory access.

For operational AI agents, explicit business capabilities are usually easier to govern than raw database access or generic CRUD tools. Operational agents commonly need bounded answers to bounded questions: customer summary, order search, invoice status, inventory availability, or ticket context.

The architecture can fail when the model is trusted to self-restrict instead of the system enforcing restrictions.

8. Practical Implications (1/2)

Companies connecting AI agents to legacy systems should start with read-only MCP tools. Define tools around business intent, not database tables. A tool should express what the caller is allowed to do, not how the database is organized.

Filter tool discovery based on caller identity or API-key scope. Enforce authorization again at runtime. The runtime check is still required because a caller can attempt `tools/call` by name even if a tool was not returned by `tools/list`.

Use input validation, execution policy, prepared SQL, row limits, byte limits, timeouts, and output allow-lists. Avoid exposing raw SQL tools, raw schema access, broad CRUD tools, and database credentials to the AI runtime or gateway.

Use capability-scoped API keys. Rotate and revoke keys when access requirements change.

8. Practical Implications (2/2)

Keep sensitive business data out of logs. Gateway logs should be useful for operations without becoming a second data store. Monitor calls by capability or tool name, caller identity, status, error code, request identifier, and latency.

Avoid logging raw SQL, database connection strings, credentials, authorization headers, API keys, tokens, full parameter values, full result sets, stack traces, or database-driver details in provider-cloud logs.

Review MCP tool descriptions as security-sensitive content. Descriptions should help the model choose an allowed operation without disclosing internal incident notes, secret workflow names, administrative functions, credentials, or schema details.

Separate partner, support, and internal-admin access profiles. The safest catalog is the one the caller actually needs.

9. Limitations and Future Work (1/2)

This evaluation is illustrative and is not a substitute for a full security audit, formal threat model, penetration test, or red-team assessment. The experiment does not implement a full MCP server and does not cover every MCP client.

The comparison does not cover every database engine, driver, ORM, API gateway, agent framework, identity provider, deployment topology, or logging platform. It assumes capability definitions and tool descriptions are reviewed and correctly implemented.

The paper focuses primarily on read-oriented access. Write operations require additional controls, approval flows, idempotency, transaction design, change logs, rollback strategy, and stronger auditing.

Human operational processes, incident response, access review workflows, and performance or latency benchmarks are not fully evaluated.

9. Limitations and Future Work (2/2)

Future work should evaluate real MCP clients, real-world prompt-injection corpora, multiple database engines, different API gateways, red-team testing, formal threat modeling, and write-operation safety.

Additional work should also examine how capability catalogs are reviewed over time, how drift is detected between `capability.yaml`, OpenAPI descriptions, MCP tool definitions, gateway authorization policy, and agent-side execution policy.

Capability-based MCP exposure is not the best fit when the tool definitions would become so broad that they effectively recreate database access, or when human analysts need exploratory access with mature governance controls.

The conclusion should therefore remain conditional: this design can reduce exposed authority and improve auditability when the capability model, authorization layer, agent implementation, and operational controls are correct.

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 GenAI Security Project, "OWASP Top 10 for Large Language Model Applications", https://owasp.org/www-project-top-10-for-large-language-model-applications/
4. OWASP API Security Project, https://owasp.org/www-project-api-security/
5. NIST, "SP 800-207: Zero Trust Architecture", https://www.nist.gov/publications/zero-trust-architecture-0
6. OpenAPI Initiative, "API Endpoints", https://learn.openapis.org/specification/paths.html
7. PostgreSQL Documentation, "Row Security Policies", https://www.postgresql.org/docs/17/ddl-rowsecurity.html

These references are used for protocol terminology, authorization framing, LLM and API security risks, zero-trust principles, API catalog terminology, and database row-level security background. They do not certify the illustrative evaluation in this paper.

Finally

The analysis does not claim that least-privilege MCP design is secure in an absolute sense.

It shows a narrower authority model for operational AI access to legacy systems. The design reduces exposed authority, limits discoverable tools, narrows the callable surface, can reduce prompt-injection blast radius, makes authorization boundaries explicit, and improves auditability by business operation.

Those benefits depend on correct capability design, tool filtering, runtime authorization, output controls, logging discipline, key management, and operation of the on-prem agent environment.

Published by: ViewLegacy LLC.
Author: ASAHI MASUDA
Publication date: June 5, 2026