ViewLegacy.

Architecture Design for Limiting the Blast Radius of Runaway AI Agents

Technical white paper

Published by: ViewLegacy LLC.
Author: ASAHI MASUDA
Publication date: August 3, 2026

This paper asks a narrow systems question: when an AI agent makes a bad decision, how can the surrounding architecture ensure that the agent can cause only a limited and auditable amount of damage?

1. Executive Summary (1/2)

AI-agent behavior cannot be assumed to remain correct under every prompt, context, failure, retry, or runtime compromise. "Runaway" behavior in this paper includes repeated calls, incorrect tools or parameters, unauthorized attempts, excessive retrieval, prompt-injection-driven misuse, unintended writes, and chained misuse across tools. It does not imply that the model is conscious, autonomous, or malicious.

This paper compares six containment patterns: direct database credentials, unrestricted SQL execution, broad CRUD or internal APIs, a large unfiltered MCP tool catalog, role-filtered tools with runtime authorization, and capability-based execution with gateway and on-premise policy enforcement.

The primary conclusion is that prompts and model alignment are useful guidance but are not deterministic security boundaries. Architecture can reduce exposed authority by replacing raw system access with predefined business operations and enforcing authorization, validation, resource limits, output filtering, rate control, and audit metadata outside the model.

1. Executive Summary (2/2)

Capability-based containment can narrow the executable surface and limit the maximum impact of one incorrect or adversarially influenced decision. The strongest version combines caller-scoped discovery, per-call authorization at a gateway, an independent on-premise capability allow-list, deterministic input and output contracts, prepared statements, a read-only database account, row and byte limits, timeouts, and rate controls.

The conclusion is bounded. Read-only access materially reduces destructive risk, but confidentiality, inference, cost, and availability risks remain. A compromised on-premise host may expose credentials and capability logic. An over-permissioned key, a broad capability, payload-bearing logs, or an unsafe write capability can defeat the intended boundary.

Capability-based access is not always the best fit. Governed warehouses or BI semantic layers are often better for broad analytics; privileged humans may need controlled administrative consoles; low-latency local automation may require direct, tightly governed execution. Containment reduces risk; it does not eliminate it.

2. Background (1/2): Why Agent Integration Changes Risk

Traditional applications follow mostly developer-authored paths. An AI agent instead selects tools and parameters dynamically, can chain outputs into later calls, and may retry after errors. A request can be technically valid while operationally inappropriate: the correct endpoint, wrong customer; a legal enum, wrong business state; or a useful query repeated hundreds of times.

Tool choice can be influenced by ambiguous user intent, incorrect reasoning, compromised context, direct prompt injection, or instructions hidden in retrieved content. OWASP describes prompt injection as input that changes model behavior in unintended ways and notes that RAG and fine-tuning do not fully mitigate it [1]. OWASP's excessive-agency category links damaging outcomes to excessive functionality, permissions, or autonomy [2].

The security question therefore shifts from "Will the agent follow policy?" to "What can the surrounding system enforce even when it does not?"

2. Background (2/2): Broad Tools Amplify a Bad Decision

A broad tool compresses many decisions into one authority grant. Database credentials expose the account's whole privilege set. Raw SQL exposes a language. Generic CRUD exposes objects and methods. A large MCP catalog exposes many named actions and their schemas. The model then becomes a de facto policy decision point even though its output is probabilistic.

Legacy systems make this harder. They may lack modern per-operation authorization, consistent validation, bounded pagination, structured audit events, or caller-aware rate limits. A database privilege model may be too coarse for a partner asking for one invoice status.

MCP tools are designed to be discoverable and model-controlled; the specification also requires servers to validate inputs, enforce access controls, rate-limit invocations, sanitize outputs, and recommends user confirmation for sensitive actions [3]. Those safeguards must be implemented by the surrounding system. A tool description or system prompt cannot substitute for them.

3. Compared Containment Architectures (1/6)

Pattern 1: AI agent with direct database credentials

The agent runtime receives a connection string or equivalent database credential. It can discover whatever schema metadata and data the account permits, submit native queries, and often maintain sessions. Parameter and output freedom are bounded mainly by the database engine and account privileges.

- **Authority and writes:** all read and write privileges granted to the account.
- **Discovery and calling:** schema-wide objects and native database operations.
- **Rate control and audit:** usually database-side; caller identity may collapse to one shared account.
- **Expected blast radius:** the entire granted database authority, including destructive or cross-tenant effects if the account is broad.
- **Model dependence:** highest. The architecture relies heavily on the agent to compose appropriate queries and stop at the intended scope.

Read-only credentials reduce integrity risk, but do not inherently bound rows, bytes, joins, lock pressure, query cost, or sensitive columns.

3. Compared Containment Architectures (2/6)

Pattern 2: AI agent with unrestricted SQL execution

The agent calls a tool such as `run_sql(sql_text)`. Credentials may be hidden inside the tool host, but the SQL language remains part of the agent's executable surface. The distinction from direct credentials is operational custody, not necessarily authority.

- **Authority and writes:** all statements accepted by the wrapper and database account.
- **Discovery and calling:** one apparently small tool can expose an entire query and command language.
- **Parameter and output freedom:** extremely high unless the wrapper implements a real parser, statement policy, result filtering, and resource controls.
- **Rate control and audit:** can exist at the wrapper, but business-operation attribution is weak because logs record SQL rather than intent.
- **Expected blast radius:** close to direct database access when arbitrary SQL is accepted.
- **Model dependence:** very high; a prompt rule such as "SELECT only" is guidance, not enforcement.

Prepared statements do not make arbitrary SQL safe; they protect data parameters only when the statement shape is predefined.

3. Compared Containment Architectures (3/6)

Pattern 3: Broad CRUD APIs or broad internal APIs

Generic endpoints such as `list_table`, `update_record`, or a complete internal API catalog replace SQL with structured calls. This improves input syntax and can reuse existing identity systems, but the agent may still reach many objects, methods, fields, and workflows.

- **Authority and writes:** every exposed endpoint and permitted HTTP method.
- **Discovery and calling:** broad resource inventory, often including administrative functions.
- **Parameter and output freedom:** less than SQL, but filters, sort keys, field selectors, bulk operations, and pagination may remain open-ended.
- **Rate control and audit:** commonly available at an API gateway, with endpoint-level granularity.
- **Expected blast radius:** all exposed APIs and their reachable objects.
- **Model dependence:** medium to high unless every endpoint enforces caller-specific authorization and bounded contracts.

OWASP recommends deny-by-default function authorization with explicit grants and consistent checks across business functions [4].

3. Compared Containment Architectures (4/6)

Pattern 4: MCP server with a large unfiltered tool catalog

An MCP server can publish many tools through `tools/list`, including business, generic, or administrative operations. Tool schemas improve machine usability, but a large catalog also expands what a compromised or confused runtime can discover and attempt.

- **Authority and writes:** every callable tool; may include side effects.
- **Discovery and calling:** the catalog advertises names, descriptions, and input schemas to every caller.
- **Parameter and output freedom:** depends on each tool; one raw SQL or export tool can dominate the risk.
- **Rate control and audit:** server or gateway controlled, if implemented.
- **Expected blast radius:** the union of all published tool authorities.
- **Model dependence:** high when the server assumes the model will choose only contextually appropriate tools.

Filtering discovery reduces information exposure and accidental selection, but it is not authorization. A caller can still send `tools/call` directly.

3. Compared Containment Architectures (5/6)

Pattern 5: Role-filtered tools with runtime authorization

The server returns only tools allowed for the caller's role and repeats the authorization check on every invocation. A leaked sales key cannot call an admin tool if the mapping and enforcement are correct, even if the tool name is guessed.

- **Authority and writes:** the union of tools granted to the role.
- **Discovery and calling:** reduced to role scope; unauthorized direct calls are rejected.
- **Parameter and output freedom:** still determined by each tool. A granted export or generic search tool may be broad.
- **Rate control and audit:** normally centralized at the API or MCP server.
- **Expected blast radius:** the role's callable tools and each tool's internal freedom.
- **Model dependence:** lower for cross-role access, but may remain high inside an allowed tool.

This pattern is a meaningful improvement. It is not equivalent to constrained execution if the granted tools lack row, byte, field, timeout, or business-state limits.

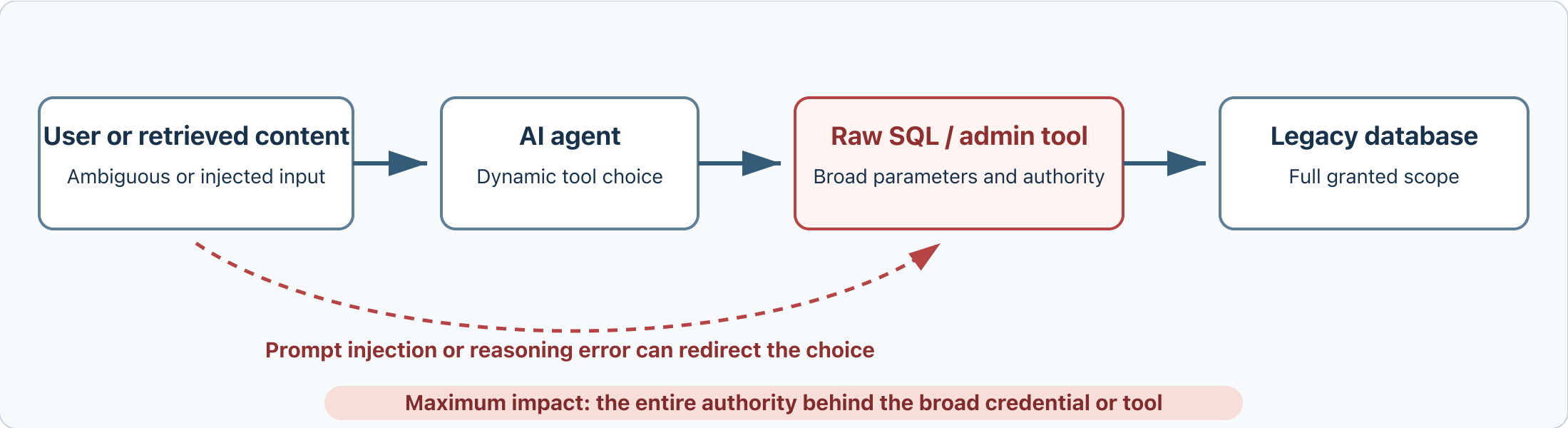
3. Compared Containment Architectures (6/6)

Pattern 6: Capability-based gateway and on-premise enforcement

The agent calls named business capabilities such as `search_orders`, not tables or SQL. A gateway authenticates the caller, applies IP and rate policy, filters discovery, checks the API-key capability scope, and routes the request. The on-premise agent independently rejects undefined capabilities, validates inputs, applies execution policy, executes predefined prepared SQL, and filters output.

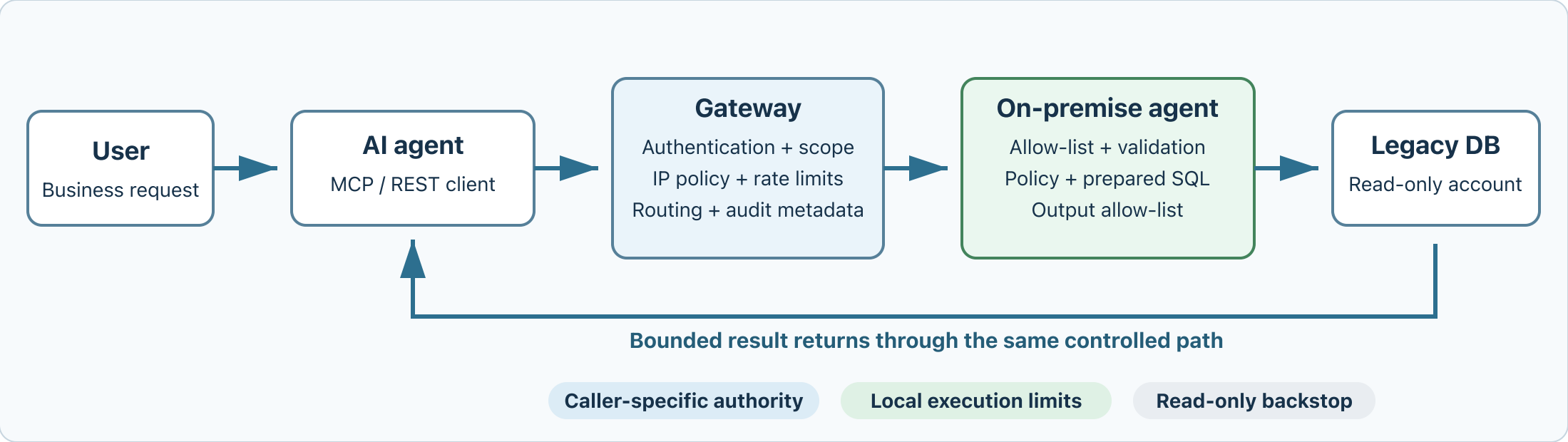
- **Authority and writes:** explicitly defined and authorized capabilities; initial scope is read-only.
- **Discovery and calling:** caller-scoped catalog plus per-call authorization.
- **Parameter and output freedom:** schema, enums, ranges, lengths, result fields, rows, bytes, and time are bounded.
- **Rate control and audit:** gateway plus agent concurrency controls; logs identify business operations without payloads.
- **Expected blast radius:** authorized capabilities within enforced limits, assuming correct design and implementation.
- **Model dependence:** lower because the decisive boundaries are external to the model.

3. Diagram 1: Broad Authority, Large Blast Radius



A single mistaken selection can cross tables, tenants, business processes, and read/write boundaries. The model is asked to self-restrict inside an authority grant that is larger than the task.

3. Diagram 2: Capability-Based Containment

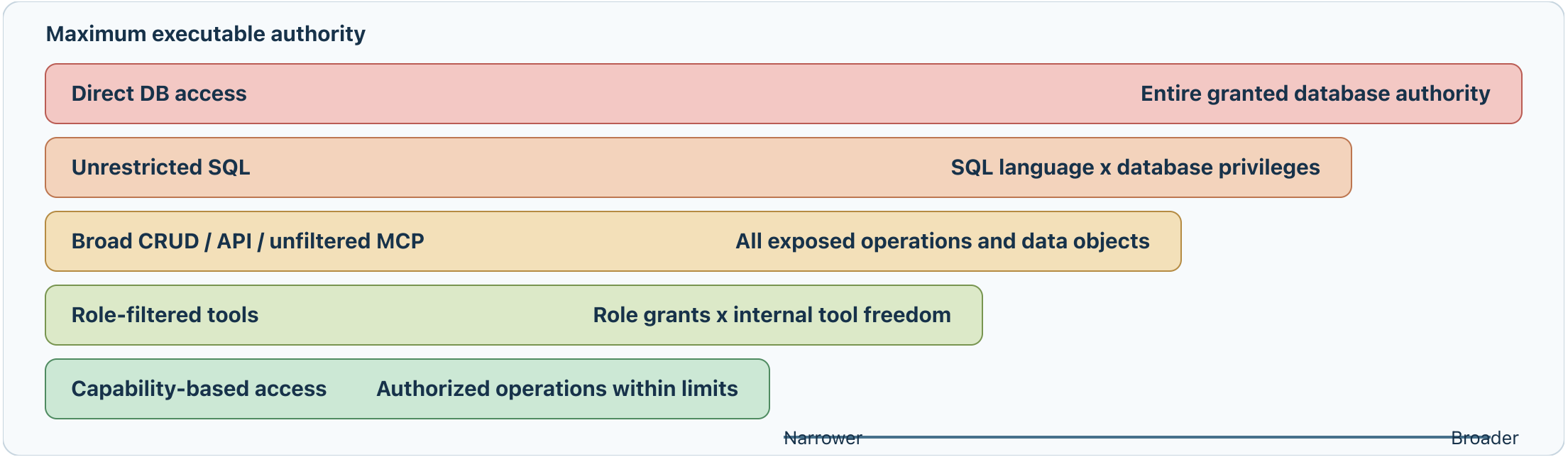


The gateway does not need raw SQL or database credentials. The on-premise boundary does not trust the gateway to invent operations: undefined capabilities are rejected locally.

3. Diagram 3: Layered Enforcement Model

No single layer is sufficient. Discovery filtering is usability and exposure reduction; runtime authorization is the access decision; agent and database controls bound what an authorized request can do.

3. Diagram 4: Maximum Impact by Architecture



This ordering concerns maximum executable authority, not the probability of failure. A compromised host or badly designed capability can still produce critical impact.

3. Comparison: Authority and Executable Surface

Pattern	Maximum executable authority	Discoverable operations	Callable operations	Parameter freedom	Output freedom	Write exposure	Model self-restraint
Direct DB credentials	DB-account privileges	Schema, metadata, DB features	Native DB operations	Very high	Very high	Account-dependent, often broad	Very high
Unrestricted SQL	Accepted SQL x DB privileges	SQL tool plus inferable schema	Arbitrary accepted statements	Very high	Very high	High unless parsed and enforced	Very high
Broad CRUD / internal APIs	All exposed endpoints and objects	Broad API inventory	Exposed methods	High	High	Often create/update/delete	High
Unfiltered MCP catalog	Union of every published tool	Full tools/list catalog	Every tool without per-caller auth	Tool-dependent	Tool-dependent	Catalog-dependent	High
Role-filtered + runtime auth	Role-granted tools	Role-filtered	Role-authorized	Tool-dependent	Tool-dependent	Role/tool-dependent	Medium
Capability-based containment	Authorized capabilities within limits	Caller-scoped capabilities	Authorized and locally defined only	Low to bounded	Explicit fields and limits	None in initial read-only scope	Lower

The capability pattern's advantage disappears if a capability becomes a generic query language, unrestricted export, or broad administrative wrapper.

3. Comparison: Enforcement, Operations, and Failure Scope

Pattern	Rate-control location	Audit granularity	Revocation	Failure isolation	Legacy compatibility	Operational complexity	Expected blast radius
Direct DB credentials	DB / network	Shared DB identity, SQL	Credential rotation	Weak unless separate accounts/DBs	Native but invasive	Low initial, high governance	Entire granted DB scope
Unrestricted SQL	Tool host / DB	SQL statement	Tool token + DB credential	Weak	Broad DB support	Medium	Near direct DB scope
Broad CRUD / internal APIs	API gateway	Endpoint/object	Token/key rotation	Medium	Requires API coverage	Medium to high	All exposed APIs
Unfiltered MCP catalog	MCP server/gateway	Tool name	Token/key rotation	Tool-dependent	Adapter required	Medium	All published tools
Role-filtered + runtime auth	MCP/API gateway	Caller + tool	Role/key update	Better across roles	Adapter required	High	Role-granted tools
Capability-based containment	Gateway + agent queues/timeouts	Caller + business capability + status	Scoped key revoke/rotate	Stronger per capability; host remains shared	Predefined SQL suits legacy DBs	Highest design effort	Authorized capability limits

Operational complexity is real: explicit policy, key scopes, review, monitoring, and local deployment must remain correct over time.

4. Evaluation Criteria (1/2): Authority and Data Movement

The evaluation asks what a misled or compromised agent can execute, not merely what documentation says it should execute.

Axis	Question used in the evaluation
Maximum executable authority	What is the largest action set reachable with current credentials and tools?
Destructive-operation exposure	Can the agent create, update, delete, execute DDL, or invoke side effects?
Data-exfiltration surface	Which records, fields, tenants, joins, and exports can be retrieved?
Prompt-injection blast radius	If tool selection is hijacked, what maximum authority is available?
Parameter constraint strength	Are types, enums, ranges, lengths, formats, and unknown fields enforced?
Output minimization	Are returned fields explicitly allow-listed for the business purpose?
Row and byte limits	Are volume limits enforced after execution as well as requested in parameters?
Timeout enforcement	Can one request occupy database, agent, or network resources indefinitely?

These axes cover confidentiality, integrity, and availability. Read-only is a major integrity control, not a complete security posture.

4. Evaluation Criteria (2/2): Identity and Operations

Axis	Question used in the evaluation
Rate limiting and burst control	Are repeated calls bounded by caller/IP and by downstream capacity?
Capability-scoped authorization	Does each identity receive explicit per-operation grants with deny-all default?
Tool discovery filtering	Does tools/list reveal only the caller's authorized capabilities?
Auditability by business operation	Can operators attribute status and latency to caller and named intent without payloads?
Revocation and key rotation	Can one caller be disabled or narrowed without changing DB credentials for all users?
Failure isolation	Does compromise or overload of one layer bypass controls in another?
Legacy-system compatibility	Can controls be added without changing the legacy application's source code?
Operational complexity	What policy design, deployment, testing, review, and incident-response work is required?
Dependence on AI self-restraint	Which decisions are enforced only by prompts or model behavior?

The assessment distinguishes policy guidance from enforcement evidence. A tool description may discourage misuse; only a server-side decision can reject it.

5. Threat Model and Evaluation Method (1/5): Actors

The threat model includes both ordinary failure and compromise:

- a legitimate user with an ambiguous request
- an AI agent selecting the wrong tool or parameters
- direct or indirect prompt injection embedded in retrieved content
- a compromised agent runtime that ignores user intent
- a leaked API key used from an unauthorized context
- a compromised gateway attempting to bypass caller policy
- a compromised on-premise agent host with access to local secrets and code
- a malicious or mistaken administrator changing key scope or capability policy
- a faulty capability definition that is broader than its business name implies

The model does not assume that every actor is malicious. The same containment boundary should limit a retry bug, a hallucinated parameter, a stolen key, and an injected instruction where their attempted operations are equivalent.

5. Threat Model and Evaluation Method (2/5): Trust Boundaries

The gateway and on-premise agent are treated as potentially compromisable components, not trusted absolutes. Their consequences differ:

- **Model-level guidance:** shapes choice and user experience; it is not a deterministic policy boundary.
- **Gateway enforcement:** authenticates, scopes capabilities, filters discovery, applies IP/rate policy, routes, and records non-payload metadata.
- **Agent-side enforcement:** owns the operation vocabulary, input contract, raw SQL, prepared execution, limits, output contract, and local detailed errors.
- **Database permissions:** provide a final least-privilege backstop, especially a read-only account.
- **Operational monitoring:** detects denials, spikes, errors, latency, policy changes, and connection anomalies without collecting business payloads.

Gateway compromise should not create an undefined operation on the agent. Agent-host compromise is more severe because local database credentials and executable policy are in that trust zone.

5. Threat Model and Evaluation Method (3/5): Illustrative System

The simulation models a fictional legacy order-management system. Defined operations are `get_customer_summary`, `search_orders`, `get_invoice_status`, `check_inventory`, `export_order_report`, `update_order_status`, `delete_order`, and `run_sql`. Key profiles are `sales_partner`, `support_agent`, `internal_operator`, and `admin`.

For each of the six architectures, a control vector states whether it provides filtered discovery, runtime authorization, an on-premise capability allow-list, input validation, read-only enforcement, row/byte/time limits, an output allow-list, rate limits, payload-free logs, and agent-side enforcement.

The local evaluator validates that exactly 15 scenarios have one outcome for every pattern, then generates the scenario and control matrices. Outcomes are qualitative: **Critical**, **High risk**, **Medium risk**, **Lower risk**, **Blocked by design**, **Blocked if correctly implemented**, **Depends on implementation**, **Allowed**, or **Out of scope**.

This exercise evaluates expected maximum impact under stated assumptions. It is illustrative, not a formal proof, complete penetration test, or probability estimate.

5. Threat Model and Evaluation Method (4/5): Scenarios S1-S8

ID	Scenario	Primary property tested
S1	Prompt injection asks the agent to call `run_sql`	Removal of raw query language from executable surface
S2	Prompt injection asks the agent to delete an order	Read-only scope and write exclusion
S3	Agent calls an allowed search tool 500 times	Rate, burst, concurrency, and downstream load controls
S4	Agent requests 100,000 rows	Numeric validation plus enforced row/byte ceilings
S5	Agent requests output fields not needed for the task	Output allow-list and data minimization
S6	Sales-partner agent selects the wrong capability	Discovery filtering and runtime authorization
S7	Agent passes an invalid enum value	Deterministic schema validation
S8	Leaked sales-partner key calls an admin capability	Capability-scoped credentials and deny-all default

The evaluator interprets a broad credential in S8 as the equivalent access token for that pattern. The intent is to compare credential blast radius, not protocol syntax.

5. Threat Model and Evaluation Method (5/5): Scenarios S9-S15

ID	Scenario	Primary property tested
S9	Compromised agent runtime enumerates all tools	Caller-filtered `tools/list` and inventory exposure
S10	Compromised gateway invokes an undefined capability	Independent agent-side allow-list
S11	On-premise agent host is compromised	Credential, SQL, runtime, and endpoint trust limits
S12	Legitimate caller makes an allowed read with valid parameters	Preservation of intended function
S13	Write capability is added without adequate review	Capability design and change governance
S14	Retry loop causes availability degradation	Rate, timeout, queue, and concurrency limits
S15	Logs accidentally include sensitive payload data	Logging boundary and secondary disclosure

The scenario set deliberately includes architectural defeat cases. A useful containment analysis must show where the preferred pattern still produces high or critical residual impact.

5. Illustrative Policy Example (1/2)

```yaml
capabilities:
  search_orders:
    description: "Search recent orders for an authorized customer"
    sql: |-
      SELECT order_id, status, ordered_at
      FROM orders
      WHERE customer_id = :customer_id AND status = :status
      LIMIT :limit
    params:
      customer_id: { type: string, required: true, maxLength: 64 }
      status:
        type: string
        enum: ["open", "shipped", "cancelled"]
      limit: { type: integer, default: 50, minimum: 1, maximum: 200 }
```

The operation name expresses business intent. The fixed statement shape and prepared parameters remove SQL grammar from the caller's authority. The enum, length, and numeric range reject malformed or excessive input before execution.

5. Illustrative Policy Example (2/2)

```
policy:
  readonly: true
  timeout: 5s
  max_rows: 200
  max_bytes: 256KB
result:
  order_id: { type: string }
  status: { type: string }
  ordered_at: { type: string }
```

`readonly` excludes write statements; the database account should independently be read-only. `timeout` bounds execution occupancy. `max_rows` and `max_bytes` cap retrieval even if query estimates or caller parameters are wrong. `result` is an output allow-list, so unapproved columns are not returned.

This fragment is illustrative, not a complete production configuration. It omits caller-to-capability grants, IP policy, rate settings, secret storage, transport security, schema migration, monitoring, and capability-review workflow.

6. Evaluation Results: Containment Pattern Comparison

Pattern	Runtime constraints	Write exposure	Audit granularity	Expected blast radius	Main residual risk
Direct DB credentials	DB-native only unless separately built	Full account scope	Shared account and SQL	Entire granted database authority	Credential or runtime compromise becomes DB compromise
Unrestricted SQL tool	Wrapper-dependent; often weak	All accepted statements	Tool + SQL, weak business intent	Near the DB account's authority	One language-like tool recreates broad access
Broad CRUD / APIs	Endpoint-dependent	Broad create/update/delete	Endpoint/object	All exposed APIs and reachable data	Authorization drift, bulk methods, excessive fields
Unfiltered MCP catalog	Tool-dependent	Any published write tool	Tool name	Union of published tools	Discovery and selection of unrelated/admin tools
Role-filtered + runtime auth	Strong between roles, variable inside tools	Role-granted writes	Caller + tool	Role grant x tool internal freedom	Broad authorized tool and parameter/output freedom
Capability-based containment	Capability, schema, rows, bytes, time, output	None in evaluated read-only scope	Caller + business operation + result metadata	Authorized capabilities within limits	Bad capability design, host compromise, availability, leakage

The result is conditional: "within limits" assumes the allow-list and every enforcement layer are correct.

6. Scenario Outcomes (1/3): Direct Misuse and Volume

Scenario	Direct DB	Raw SQL	Broad CRUD/API	Unfiltered MCP	Role-filtered	Capability-based
S1 `run_sql` injection	Critical	Critical	Blocked by design	High risk	Blocked if correct	Blocked by design
S2 delete injection	Critical	Critical	High risk	High risk	Blocked if correct	Blocked by design
S3 500-call loop	High risk	High risk	High risk	High risk	Medium risk	Lower risk
S4 100,000 rows	Critical	Critical	High risk	High risk	High risk	Blocked if correct
S5 extra fields	Critical	Critical	High risk	High risk	Medium risk	Blocked if correct

"Blocked if correct" is not a guarantee. It means a named deterministic control is expected to reject the attempt and must be tested. "Lower risk" means impact is bounded more tightly, not absent: allowed calls still consume resources and may return sensitive data within the authorized scope.

6. Scenario Outcomes (2/3): Scope and Compromise

Scenario	Direct DB	Raw SQL	Broad CRUD/API	Unfiltered MCP	Role-filtered	Capability-based
S6 wrong capability	High risk	High risk	Medium risk	High risk	Blocked if correct	Blocked if correct
S7 invalid enum	High risk	High risk	Depends on impl.	Depends on impl.	Depends on impl.	Blocked if correct
S8 leaked sales key -> admin	Critical	Critical	High risk	High risk	Blocked if correct	Blocked if correct
S9 enumerate all tools	Critical	High risk	High risk	High risk	Lower risk	Lower risk
S10 compromised gateway -> undefined	Out of scope	Out of scope	Depends on impl.	Depends on impl.	High risk	Blocked if correct

S10 distinguishes gateway authorization from agent-side enforcement. A capability architecture does not let the gateway define SQL or invent a new capability. The local agent rejects names absent from its loaded policy, limiting the gateway compromise to valid local capabilities and their limits.

6. Scenario Outcomes (3/3): Residual and Operational Risk

Scenario	Direct DB	Raw SQL	Broad CRUD/API	Unfiltered MCP	Role-filtered	Capability-based
S11 on-premise host compromised	Critical	Critical	Critical	Critical	Critical	Critical
S12 valid allowed read	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed
S13 unsafe write added	High risk	High risk	High risk	High risk	High risk	High risk
S14 retry-loop availability	High risk	High risk	High risk	High risk	Medium risk	Lower risk
S15 payload captured in logs	High risk	High risk	High risk	High risk	High risk	High risk

The preferred architecture intentionally does not "win" every scenario. Host compromise crosses the local secret boundary. An unsafe capability changes the security boundary itself. Payload logging creates a durable secondary data exposure. These require host hardening, change control, log schema enforcement, and incident response beyond tool scoping.

6. Blast-Radius Comparison Matrix

Failure or misuse	Direct DB / raw SQL	Broad API / unfiltered MCP	Role-filtered tools	Capability-based containment
Wrong tool	Query language or broad account remains reachable	Any published endpoint/tool may run	Limited to role grants	Limited to scoped, locally defined capabilities
Wrong parameter	Arbitrary query predicates and values	Endpoint schema-dependent	Tool-dependent	Type, enum, range, length, format enforced
Excessive retrieval	Database-scale reads possible	Pagination/export dependent	Tool-dependent	Row + byte + output-field ceilings
Unintended write	Account/tool may write broadly	Published write methods reachable	Role-granted writes reachable	Excluded in initial read-only scope; DB account backstop
Repeated call	Load bounded mainly by DB	Gateway controls may help	Rate control reduces burst	Rate + timeout + bounded concurrency/queue reduce amplification
Leaked credential	Full credential authority	Token's broad catalog scope	Role scope	Explicit capability scope; unspecified means zero
Gateway compromise	Not applicable	Backend authority depends on gateway trust	May reach broad backend tools	Agent still rejects undefined names and applies local limits
Agent-host compromise	Equivalent DB/runtime compromise	Equivalent adapter/runtime compromise	Equivalent adapter/runtime compromise	Critical residual: DB credentials and policy may be exposed

This matrix describes maximum reachable impact, not likelihood or control assurance.

6. Control-to-Risk Mapping (1/2)

Control	Risk reduced	What it does not solve	Failure if misconfigured
Capability allow-list	Undefined or generic operation attempts	Broad defined capabilities; host compromise	Allow-all or wildcard recreates broad access
API-key capability scope	Leaked-key and cross-role authority	Abuse inside allowed capabilities	Over-permissioned key widens blast radius
Zero permission by default	Accidental grants and configuration omissions	Explicit bad grants	Default allow silently exposes new capabilities
Filtered `tools/list`	Inventory disclosure and wrong-tool selection	Direct guessed calls	Unfiltered list reveals admin and write tools
Runtime authorization	Unauthorized invocation even if name is known	Bad authorized action	Missing check makes discovery filter cosmetic
Input validation	Invalid types, enums, ranges, lengths, formats	Semantically wrong but valid input	Coercion or unknown fields bypass intent
Prepared statements	SQL injection through data parameters	Unsafe fixed SQL; compromised host	String concatenation restores injection surface
Read-only mode + DB account	Destructive SQL and unintended writes	Confidentiality, inference, availability	Writable account defeats application-only claim

Controls are compositional. The database account backs up statement policy; runtime authorization backs up discovery filtering; explicit key scope backs up user-facing intent.

6. Control-to-Risk Mapping (2/2)

Control	Risk reduced	What it does not solve	Failure if misconfigured
Row + byte limits	Bulk extraction and oversized responses	Repeated small reads; sensitive allowed fields	High limits make the cap nominal
Timeout	Long query/resource occupancy	Many short expensive calls	Missing cancellation leaves DB work running
Output allow-list	Unnecessary column disclosure	Sensitive values in approved fields; inference	Permissive result schema leaks hidden columns
Rate + burst limits	Retry loops and high-frequency abuse	Distributed sources; slow expensive calls	Disabled/shared buckets permit amplification
IP restrictions	Use from unexpected network origins	Compromised allowed host; proxy mistakes	Trusted-header error enables spoofing
Payload-free gateway logs	Secondary cloud/log disclosure	Sensitive metadata and local logs	Debug payload logging breaks data-minimization claim
Local detailed errors	Troubleshooting without cloud leakage	Local host/log compromise	Raw errors sent upstream reveal SQL or values
Single-agent connection	Competing or duplicate connector sessions	Malicious live agent; per-call abuse	Weak agent auth permits connection takeover
Explicit capability review	Unsafe scope and write introduction	Undetected implementation flaws	Unreviewed changes redefine the boundary

Single-agent connection control is operational isolation, not an authorization substitute. Sensitive changes still need authenticated rollout and monitoring.

6. Risk Movement: Controls Change Location, Not Reality

Risk	Broad access location	Capability-based location	Residual control required
SQL and credentials	Agent runtime or broad tool host	On-premise agent only	Host hardening, secret protection, read-only DB account
Authorization	Model choice or coarse credential	Gateway key scope plus per-call check	Deny-all defaults, review, revocation tests
Operation definition	SQL/API/tool catalog	`capability.yaml`	Security-sensitive code review and linting
Data minimization	Agent query or broad response	Agent-side result allow-list	Field classification and contract tests
Availability	Legacy DB absorbs broad workloads	Gateway and agent share control	Rate benchmarking, concurrency budgets, monitoring
Error detail	Tool/API responses and central logs	Local detailed error log	Local access control, rotation, retention, incident handling
Gateway compromise	May expose backend credentials/logic	Cannot define SQL; can still invoke valid capabilities	Agent allow-list, limits, transport auth, anomaly response
Agent-host compromise	Adapter or DB boundary	Highest-impact local trust zone	OS/network hardening, EDR, least privilege, recovery

The architecture moves trust from model behavior toward explicit controls, but it also concentrates critical trust in capability design and the on-premise host.

6. Results Interpretation and Residual Risk

The illustrative matrix shows the largest improvement for tool-language removal, caller scope, input/output bounding, and independent enforcement. Prompt injection can still influence tool choice, but its maximum executable effect is reduced when dangerous operations do not exist and allowed operations have narrow contracts.

Availability remains probabilistic. Rate limits, timeouts, a bounded request queue, and a single active agent connection reduce amplification and improve failure isolation. They cannot ensure that a permitted query is cheap, prevent all distributed abuse, or guarantee that a legacy database remains available under legitimate bursts.

Confidentiality also remains. Repeated bounded reads can accumulate data; approved fields can still be sensitive; response timing and existence checks may permit inference. Correlation across several capabilities can reveal more than any one response. Monitoring, caller-specific budgets, semantic authorization, and periodic capability review remain necessary.

7. Discussion (1/4): Governance Versus Flexibility

A narrow tool set is easier to reason about, test, revoke, and audit, but it is less flexible. Capability quality determines the actual boundary. `get_invoice_status(invoice_id)` can be governable; `query_any_business_data(query)` may simply recreate raw access under a safer-sounding name.

Role filtering and capability scoping solve different problems. Role filtering reduces which tools are visible and callable. Capability constraints bound what an authorized tool can do. Both are needed because an agent can misuse an allowed tool without crossing a role boundary.

Output minimization is equally important. Input validation may ensure a request is well-formed while a broad response returns extra personal, financial, or commercial fields. The result contract should be a positive allow-list and should fail safely when the query and contract disagree.

7. Discussion (2/4): Read-Only and Availability

Read-only mode prevents many destructive outcomes: deletion, unintended state transitions, schema changes, and certain ransomware-like effects. It does not prevent data leakage, inference, expensive scans, lock contention caused by reads, credential theft, or service exhaustion. "Read-only" must therefore be treated as one strong layer, not a complete risk conclusion.

Rate limiting can reduce loops and repeated-call amplification, and OWASP recommends limits on interaction frequency, records returned, payload size, and execution time [5]. Yet strict limits can reject legitimate seasonal bursts or incident-response workflows. Limits should be per caller and operation where practical, tied to legacy capacity, and tested rather than copied from generic defaults.

Asynchronous work may require job identity, quotas, cancellation, idempotency, and durable status. Synchronous request limits alone do not safely govern long-running exports or multi-step workflows.

7. Discussion (3/4): Compromise Boundaries and Logs

Gateway compromise and agent-host compromise have different consequences. A gateway can observe transient traffic and invoke valid capabilities; it should not possess SQL or DB credentials, and the agent should reject undefined operations. A compromised on-premise host may obtain the database credential, raw SQL, capability definitions, detailed local errors, and live results. That is a critical residual risk.

Logs can quietly defeat containment. Auditability requires capability name, caller identity, status, error code, timestamp, and latency. It does not require request parameters or response bodies. OWASP's logging guidance recommends excluding access tokens, database connection strings, secrets, and sensitive personal or commercial data [6].

Detailed database errors should remain local and access-controlled; upstream messages should use stable public error codes without SQL text, credentials, driver detail, or input values.

7. Discussion (4/4): Writes and Alternative Architectures

Writes require stronger safeguards than reads: change authorization, human approval for high-impact actions, idempotency keys, transaction boundaries, concurrency control, rollback or compensation, and evidence-rich audit trails. A retryable `update_order_status` must not apply the transition twice. A delete may need to be replaced by a reversible workflow.

Capability-based containment is usually a good fit for operational agents performing narrow, repeatable business actions against legacy systems. It may not be the best fit when:

- analysts need ad hoc exploration or broad joins; a governed warehouse or BI semantic layer may be better
- batch analytics or ML needs large historical datasets
- human administrators require broad interactive control through a hardened admin console
- capabilities become so broad that they recreate raw system access
- low-latency local automation needs direct execution inside a tightly governed environment

Architecture should follow the workload, not force every workload through one access pattern.

8. Practical Architecture Guidance (1/3): Define Authority First

Assume the agent will eventually make an incorrect or adversarially influenced decision. Before exposing a tool, define the maximum acceptable impact in confidentiality, integrity, availability, cost, and operational scope.

Start with read-only capabilities. Define them around business intent, not tables: `get_invoice_status` instead of `select_invoice_rows`; `check_inventory` instead of generic inventory CRUD. Exclude raw SQL, raw schema access, database credentials, bulk administrative tools, and write functions that are not essential.

Use deny-all defaults. An API key with no explicit capability list should have zero permissions. Scope every key to named capabilities, filter OpenAPI and `tools/list` output to that scope, and repeat authorization on every invocation. Maintain tested procedures to revoke and rotate credentials without rotating a shared database account for every caller.

8. Practical Architecture Guidance (2/3): Enforce Every Contract

Validate every input deterministically at the external execution boundary. Reject unknown fields; constrain types, enums, numeric ranges, string lengths, identifiers, dates, and formats. Use predefined SQL with prepared parameters, and run it through a database account whose privileges match the intended read-only scope.

Enforce timeouts, maximum rows, maximum bytes, bounded concurrency, and bounded queues. Do not rely solely on a caller-provided `limit`. Filter the response through an explicit output allow-list. Test both missing required columns and unexpected extra columns.

Apply caller/IP rate and burst controls at the gateway, with trusted-proxy handling that does not accept spoofed forwarding headers. Use one authenticated active agent connection where the deployment model requires it, but do not confuse connection exclusivity with request authorization or host integrity.

8. Practical Architecture Guidance (3/3): Operate the Boundary

Log operation metadata: request ID, timestamp, protocol, capability name, caller/API-key identity, status, public error code, and latency. Do not log business parameters, result payloads, authorization headers, SQL, DB credentials, secrets, or raw database errors. Separate operational metadata from business data and protect log access and retention.

Treat every capability change as security-sensitive. Review business purpose, caller groups, statement shape, write effects, input constraints, result fields, worst-case rows/bytes/time, query plan, concurrency impact, and rollback. For writes, add approval, idempotency, state-transition rules, transaction design, and stronger audits.

Before production, test prompt injection, wrong-tool calls, invalid values, maximum boundaries, repeated calls, retries, rate-limit behavior, revocation, gateway compromise assumptions, agent disconnects, and sensitive-data absence from responses and logs. Monitoring should alert on denial spikes, rate-limit events, unusual capability use, latency shifts, and policy changes.

9. Limitations and Future Work (1/2)

This evaluation is illustrative and is not a substitute for a security audit, formal threat model, penetration test, or production capacity benchmark. The local simulation compares stated architecture properties; it does not implement every MCP client, agent framework, database engine, API gateway, transport, or legacy system.

The paper does not prove that prompt injection can be completely prevented. It assumes deterministic controls are correctly implemented, capability definitions are correctly scoped and reviewed, identity mapping is sound, and policy cannot be bypassed through another path. It focuses primarily on read-oriented operations.

Availability attacks and high-volume abuse were not benchmarked. Human operational processes, privileged administrator behavior, supply-chain compromise, side channels, cross-capability inference, and data aggregation over long periods were not fully evaluated. The on-premise operating system, network, endpoint protection, database, and backup security remain critical.

9. Limitations and Future Work (2/2)

Write operations require additional evaluation of approvals, idempotency, transaction design, rollback or compensation, concurrency controls, change authorization, business-state invariants, and stronger audit trails. A read-oriented result should not be generalized to destructive or financial workflows.

Future work should include:

- real MCP clients and representative agent frameworks
- real prompt-injection datasets and multi-step tool-chain evaluation
- red-team exercises, formal threat modeling, and capability-review automation
- chaos testing, disconnect recovery, retry-loop tests, and rate-limit benchmarking
- accumulated-data and inference tests across repeated bounded calls
- write-operation safety, approvals, idempotency, concurrency, and rollback
- fuzzing of policy parsing, input schemas, result contracts, and error handling
- measured legacy database load under allowed and adversarial request mixes

Evidence from these tests should update both capability limits and operational runbooks.

10. References (1/2)

1. OWASP GenAI Security Project. **LLM01:2025 Prompt Injection.** Describes direct and indirect prompt injection and the limits of prompt-only mitigation.
   https://genai.owasp.org/llmrisk/llm01-prompt-injection/

2. OWASP GenAI Security Project. **LLM06:2025 Excessive Agency.** Identifies excessive functionality, permissions, and autonomy as sources of confidentiality, integrity, and availability impact.
   https://genai.owasp.org/llmrisk/llm062025-excessive-agency/

3. Model Context Protocol. **Tools - Specification 2025-11-25.** Defines `tools/list`, `tools/call`, tool schemas, and security considerations including input validation, access control, rate limiting, output sanitization, timeouts, confirmation, and audit logging.
   https://modelcontextprotocol.io/specification/2025-11-25/server/tools

4. OWASP API Security Project. **API5:2023 Broken Function Level Authorization.** Recommends deny-by-default function access with explicit grants and consistent authorization checks.
   https://owasp.org/API-Security/editions/2023/en/0xa5-broken-function-level-authorization/

10. References (2/2)

5. OWASP API Security Project. **API4:2023 Unrestricted Resource Consumption.** Covers limits on execution time, payload size, records per response, interaction frequency, and server-side parameter validation.
   https://owasp.org/API-Security/editions/2023/en/0xa4-unrestricted-resource-consumption/

6. OWASP Cheat Sheet Series. **Logging Cheat Sheet.** Provides guidance on security-event logging and exclusion or protection of tokens, credentials, connection strings, personal data, and commercial information.
   https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html

7. NIST. **Artificial Intelligence Risk Management Framework (AI RMF 1.0), NIST AI 100-1.** A voluntary, use-case-agnostic framework for managing AI risks across design, deployment, use, and evaluation.
   https://doi.org/10.6028/NIST.AI.100-1

8. NIST. **Zero Trust Architecture, SP 800-207.** Frames access around users, assets, and resources rather than implicit trust from network location.
   https://doi.org/10.6028/NIST.SP.800-207

9. MITRE. **ATLAS Matrix for AI Systems.** A knowledge base of adversary tactics and techniques for AI-enabled systems, including prompt injection and exfiltration through agent tool invocation.
   https://atlas.mitre.org/

Finally

The practical objective is not to make an AI agent incapable of error. It is to design the surrounding system so that an error, injected instruction, leaked credential, or retry loop encounters deterministic boundaries before it becomes broad system authority.

Capability-based access can reduce the blast radius by narrowing what is discoverable, callable, parameterizable, returnable, and repeatable. Its security value comes from the composition of explicit grants, runtime checks, local policy, read-only database permissions, bounded execution, output minimization, and auditable metadata.

The boundary remains only as strong as its capabilities, configuration, implementation, hosts, and operating discipline. "Expose capabilities, not your database" is therefore an architectural constraint, not a guarantee.

Published by: ViewLegacy LLC.
Author: ASAHI MASUDA
Publication date: August 3, 2026