ViewLegacy.

Defensive Effects of Input Contracts, Execution Constraints, and Output Contracts in `capability.yaml`

Technical white paper

Published by: ViewLegacy LLC.
Author: Asahi Masuda
Publication date: September 7, 2026

1. Executive Summary (1/2)

AI agents can select wrong parameters, follow adversarial instructions, or invoke a valid operation at an unsafe scale. Model prompts can reduce mistakes, but they do not form a deterministic enforcement boundary. This paper evaluates a predefined business capability with three external controls:

- **Input contract (`params`)** narrows accepted names, types, values, ranges, lengths, patterns, and formats.
- **Execution constraints (`policy`)** bound read/write behavior, duration, row count, response bytes, and public exposure. Prepared statements and a database account with minimum privilege reinforce this layer.
- **Output contract (`result`)** allow-lists fields that may leave the on-premise agent and detects missing required fields.

The evaluation uses a deterministic in-memory model with sensitive customer columns. The same 18 scenarios run through seven configurations. It counts observable contract outcomes, not breach probabilities.

1. Executive Summary (2/2)

The full three-layer configuration handled **15 of 16 adverse scenarios** by blocking, constraining, filtering, or detecting them. Two legitimate requests succeeded. The remaining adverse scenario placed sensitive text inside an allowed `name` field, which field-name allow-listing cannot recognize.

Input-only handled 8 adverse cases, policy-only handled 6 after the independent row-bound scenarios were exercised, and result-only handled 5. Combinations were stronger because each layer controls a different stage. Counts describe this simulation only.

> **Main conclusion.** A narrow capability with all three contracts produced the best result among the seven simulated configurations. Adding capability-scoped authorization and a database read-only account provides the strongest evaluated architecture. It still does not eliminate confidentiality, availability, authorization, host-compromise, or configuration risk.

Bad SQL, broad capabilities, unsafe defaults, or sensitive allowed fields can defeat the intended boundary. Read-only policy materially reduces destructive risk but cannot prevent data disclosure or expensive reads.

2. Background (1/2)

An AI agent produces probabilistic decisions. Even without an attacker, it can choose the wrong status, copy an identifier incorrectly, or request an excessive result. Prompt injection and compromised context can also push it beyond the user's intent. OWASP recommends restricting tool access and applying least privilege as part of prompt-injection defense [4].

Generic APIs often validate JSON syntax while leaving business meaning broad. A numeric `limit` may be syntactically valid at one million. A string may be well formed yet identify a customer the caller may not access. Database permissions are usually too coarse to express a five-second timeout, a 500-row ceiling, or a three-field response for one operation.

Input checks therefore reduce accepted request space but do not establish identity, ownership, consent, or purpose. Authorization remains a separate decision.

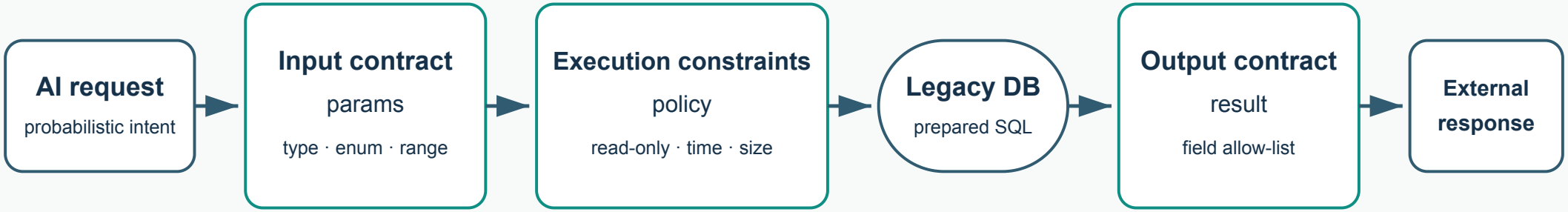
2. Background (2/2)

The capability definition acts as a machine-readable execution and interface contract. The on-premise agent loads reviewed definitions at startup, rejects undefined capabilities, validates structure before serving, and uses prepared SQL rather than model-generated SQL. Changes require a deliberate deployment or restart.

These properties move enforcement outside the model. They also make reviews and tests refer to named business operations instead of an open-ended query surface. NIST's zero-trust guidance emphasizes protecting resources and applying least privilege to visibility and access [5].

Legacy tables create a particular output problem. They commonly contain operational notes, risk scores, reset tokens, or later-added columns beside public attributes. SQL that returns more fields than intended can expose them unless a second boundary filters the response. The contract model separates query correctness from external visibility.

3. The Three Defensive Layers



Each boundary narrows a different dimension: accepted input, runtime behavior, or visible output.

The layers are sequential, but their value is not merely cumulative. They constrain different dimensions and can fail independently. A bounded input does not cap the physical query result. A timeout does not decide which fields are public. A field allow-list does not authorize a caller.

3.1 Input Contracts (1/2)

OWASP recommends server-side allow-list validation with explicit types, ranges, lengths, and finite allowed values [1]. JSON Schema provides corresponding structural keywords such as `type`, `enum`, `minimum`, `maximum`, `minLength`, and `maxLength` [6]. In a capability, these checks reject values before database execution.

Contract element	Defensive effect	Example
`type`, `required`, `default`	Stabilizes shape and missing-value behavior	`limit` must be an integer; default 50
`enum`	Restricts finite business states	`active` or `inactive` only
`minimum`, `maximum`	Rejects nonsensical or excessive magnitude	`1 <= limit <= 500`
`minLength`, `maxLength`	Bounds parsing and downstream input size	customer code up to 32 characters
`pattern`, `format`	Checks a defined lexical form	tenant code or date syntax
Unknown-field rejection	Prevents undeclared behavior switches	reject `include_internal=true`

3.1 Input Contracts (2/2)

Input contracts reduce malformed identifiers, invalid dates, oversized strings, negative or zero limits, excessive limits, and invalid enum values. An injection-like value may also fail an enum or pattern check. That check complements prepared statements; OWASP warns that validated data is not automatically safe for SQL string construction [2].

The limits are substantial:

- A valid account ID can still identify another user's account.
- Individually valid fields can form an invalid business combination.
- Regex and `format` keywords do not prove semantic truth.
- A broad enum or unsafe default makes a formally valid request dangerous.
- Some identifiers require normalization rules that a basic pattern cannot express.

Input validation does not replace authentication, capability authorization, row ownership checks, consent, or workflow approval. It answers “is this request shaped as the capability permits?” rather than “may this caller perform it on these records?”

3.2 Execution Constraints (1/2)

Execution constraints bound what happens after input acceptance. `readonly: true` rejects mutating statements at the capability layer. A database read-only account should independently deny writes if application classification fails. PostgreSQL documents transaction read-only modes while noting that “read only” is a high-level notion with engine-specific behavior [8].

`timeout`, `max_rows`, and `max_bytes` limit different resource dimensions. A timeout bounds duration, a row ceiling bounds cardinality, and a byte ceiling bounds serialized volume. OWASP API guidance recommends maximum input and response-related limits because unrestricted resource consumption can cause availability and cost impact [3].

`expose_in_openapi` keeps an internal capability out of generated discovery surfaces. It reduces discoverability, but runtime authorization must still reject calls. Discovery filtering is not an access-control substitute.

3.2 Execution Constraints (2/2)

Prepared statements keep parameter values separate from query structure and are a primary SQL-injection defense [2]. Startup linting, placeholder checks, and database validation such as `EXPLAIN` can catch malformed definitions before service begins. They cannot prove the query implements the correct business rule.

Execution controls mitigate runaway queries, excessive retrieval, accidental writes, and some injection attempts. They do not remove:

- confidentiality risk from a legitimate but broad read;
- availability risk below configured thresholds or across repeated calls;
- semantic errors in joins, filters, or tenant scoping;
- engine-specific read-only exceptions and driver cancellation behavior;
- risk from a compromised on-premise agent host.

`limit <= 500` and `max_rows <= 500` are intentionally separate. The first rejects a caller's excessive request. The second caps the actual rows returned if SQL ignores, misapplies, or later loses that parameter.

3.3 Output Contracts (1/2)

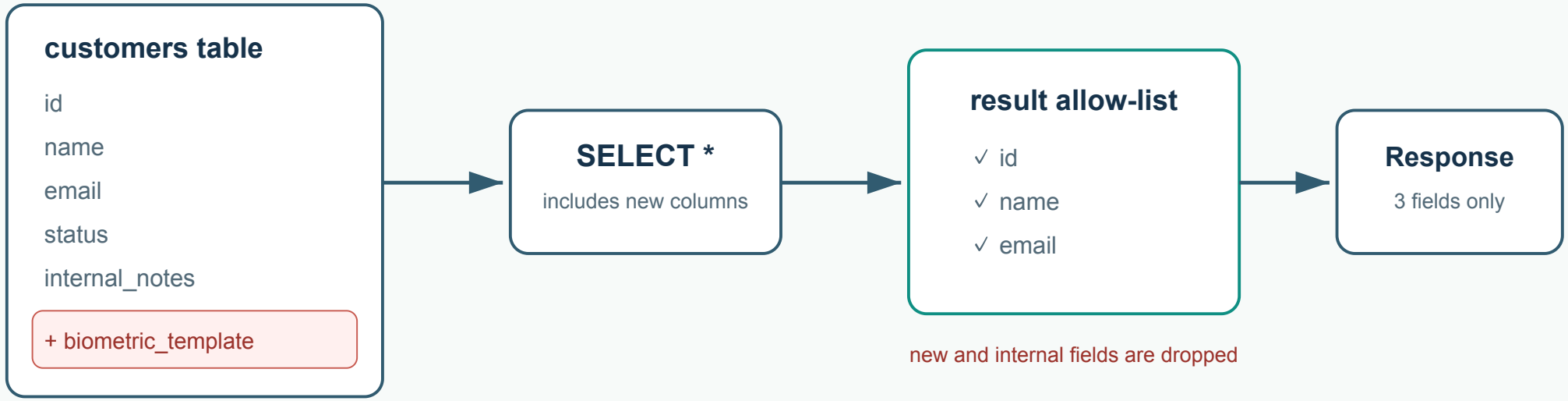
`result` defines an explicit field allow-list. The agent removes SQL fields absent from the list, fails when a required field is missing, and returns no row-object fields when the contract is absent. Therefore `SELECT *` cannot make a newly added column public by itself.

This layer provides:

- field-level minimization independent of SQL projection;
- protection against accidental `SELECT *` and later query broadening;
- fail-closed schema mismatch detection when an expected field disappears;
- a stable public interface despite internal schema evolution;
- a reviewable statement of which fields may cross the on-premise boundary.

Explicit SQL columns remain preferable because they reduce data read and processed internally. The output contract serves as an independent egress boundary, not permission to write broad SQL.

3.3 Output Contracts (2/2)



Field minimization remains stable even when the query or schema broadens.

Output allow-listing controls field names, not meaning. It cannot detect a password reset token copied into an allowed `name` field, a secret embedded in free text, or information inferred from repeated allowed queries. It also does not decide which rows a caller may view.

Row-level authorization, database row security, semantic classification, redaction, or DLP may be required when sensitivity depends on identity, ownership, or content. Output filtering also occurs after query execution, so it does not by itself reduce the database work caused by broad SQL.

4. Compared Control Configurations (1/2)

Configuration	Input freedom	Execution freedom	Output freedom	Expected blast radius	Main weakness
1. No contract	Arbitrary	Arbitrary query or API behavior	All returned fields	Unbounded by these controls	Model or caller can shape query and result
2. Input only	Declared values only	Unbounded after acceptance	All returned fields	Lower malformed-request risk	Valid requests can still run long or disclose fields
3. Policy only	Broad	Read-only, time/row/byte bounded	All fields within bounds	Runtime damage bounded	Semantically invalid input and sensitive fields remain
4. Result only	Broad	Unbounded	Allow-listed fields	Column disclosure reduced	Writes, costly reads, and wrong rows remain
5. Input + policy	Narrow	Bounded	All returned fields	Request and runtime scope reduced	Egress fields remain dependent on SQL
6. Input + result	Narrow	Unbounded	Allow-listed fields	Request and column scope reduced	Long-running and destructive execution remain
7. All three	Narrow	Bounded	Allow-listed fields	Smallest simulated boundary	Authorization, rows, semantics, host risk remain
8. All three + scoped auth + read-only DB user	Narrow and caller-scoped	Bounded with DB backstop	Allow-listed fields	Strongest architecture considered	Configuration and enforcement still determine safety

4. Compared Control Configurations (2/2)

Operational complexity rises from one open query surface to reviewed definitions, tests, deployment controls, and per-capability authorization. That burden is justified for deterministic operational actions where repeatability and bounded authority matter more than ad hoc flexibility.

Configuration 8 is the preferred production pattern in this paper, but the local experiment measures configurations 1 through 7. Authorization and database privilege need integration tests against real identity, policy, and database systems.

This model is a poor fit when analysts need flexible ad hoc SQL, workloads require broad analytical joins, or the operation must generate query structure dynamically. A governed warehouse, semantic layer, or isolated analytical environment may suit those cases better. Multi-transaction workflows and writes need approvals, idempotency, transaction design, concurrency control, and recovery beyond the three contracts.

5. Evaluation Method (1/2)

The reproducible evaluator in `src/evaluate-scenarios.mjs` uses 620 fictional customer records. The underlying model includes `internal_risk_score`, `internal_notes`, and `password_reset_token`; a schema-change scenario adds `biometric_template`. The intended capability is `search_customers(status, limit)` and the public fields are `id`, `name`, and `email`.

For each configuration, the evaluator applies enabled controls in production order:

1. Validate declared parameters and reject unknown fields.
2. Apply read-only, simulated timeout, prepared-value handling, row, and byte controls.
3. Enforce the output allow-list and detect missing required fields.

All inputs and outcomes are deterministic. The script asserts the full-contract result set before writing JSON and Markdown evidence. It requires no database or network.

5. Evaluation Method (2/2)

The 18 scenarios cover three normal or residual cases and 15 cases expected to trigger a contract response: unknown input, invalid enum, zero/negative/excessive limits, oversized strings, injection-like input, timeout, excessive rows and bytes, unexpected columns, `SELECT *`, schema expansion, missing required output, and a write attempt.

S18 deliberately puts sensitive text inside the allowed `name` field. It tests the boundary's known semantic limitation. S1 and S17 are legitimate requests and should succeed.

Outcome vocabulary:

- **Blocked:** execution or response stopped.
- **Constrained:** execution continued only within a bound or safe value-handling path.
- **Filtered:** disallowed fields removed.
- **Detected:** contract mismatch caused failure.
- **Allowed:** no enabled control changed the scenario.

This is an illustrative contract test, not a formal security proof, penetration test, performance benchmark, or incident-probability measurement.

6. Evaluation Results: Scenario Outcomes (1/3)

Scenario	No contract	Input only	Policy only	Result only	Input + policy	Input + result	Full three-layer
S1 Valid request	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed
S2 Unknown parameter	Allowed	Blocked	Allowed	Allowed	Blocked	Blocked	Blocked
S3 Invalid enum	Allowed	Blocked	Allowed	Allowed	Blocked	Blocked	Blocked
S4 Limit zero	Allowed	Blocked	Allowed	Allowed	Blocked	Blocked	Blocked
S5 Limit negative	Allowed	Blocked	Allowed	Allowed	Blocked	Blocked	Blocked
S6 Extreme limit	Allowed	Blocked	Constrained	Allowed	Blocked	Blocked	Blocked

S6 demonstrates independent bounds. Input validation rejects the requested million-row limit. Where input validation is absent, `max_rows` still constrains the query result.

6. Evaluation Results: Scenario Outcomes (2/3)

Scenario	No contract	Input only	Policy only	Result only	Input + policy	Input + result	Full three-layer
S7 Oversized string	Allowed	Blocked	Allowed	Allowed	Blocked	Blocked	Blocked
S8 Injection-like value	Allowed	Blocked	Constrained	Allowed	Blocked	Blocked	Blocked
S9 Timeout	Allowed	Allowed	Blocked	Allowed	Blocked	Allowed	Blocked
S10 Excess rows	Allowed	Blocked	Constrained	Allowed	Blocked	Blocked	Blocked
S11 Excess bytes	Allowed	Allowed	Blocked	Filtered	Blocked	Filtered	Blocked
S12 Unexpected sensitive column	Allowed	Allowed	Allowed	Filtered	Allowed	Filtered	Filtered

For S8, enum validation rejects the value before execution. Policy-only treats it as data through prepared-value handling. Neither mechanism proves that an allowed value is authorized.

6. Evaluation Results: Scenario Outcomes (3/3)

Scenario	No contract	Input only	Policy only	Result only	Input + policy	Input + result	Full three-layer
S13 SQL becomes `SELECT *`	Allowed	Allowed	Allowed	Filtered	Allowed	Filtered	Filtered
S14 Schema adds sensitive column	Allowed	Allowed	Allowed	Filtered	Allowed	Filtered	Filtered
S15 Required field missing	Allowed	Allowed	Allowed	Detected	Allowed	Detected	Detected
S16 Write under read-only	Allowed	Allowed	Blocked	Allowed	Blocked	Allowed	Blocked
S17 Legitimate intended result	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed
S18 Sensitive text in allowed field	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed

S13 and S14 show why explicit column selection and an output contract are complementary. S18 remains outside all three controls and requires content-aware or business-specific protection.

6. Evaluation Results: Counted Outcomes

Configuration	Blocked	Constrained	Filtered	Detected	Mitigated total	Allowed
No contract	0	0	0	0	**0**	18
Input only	8	0	0	0	**8**	10
Policy only	3	3	0	0	**6**	12
Result only	0	0	4	1	**5**	13
Input + policy	11	0	0	0	**11**	7
Input + result	8	0	4	1	**13**	5
Full three-layer	11	0	3	1	**15**	3

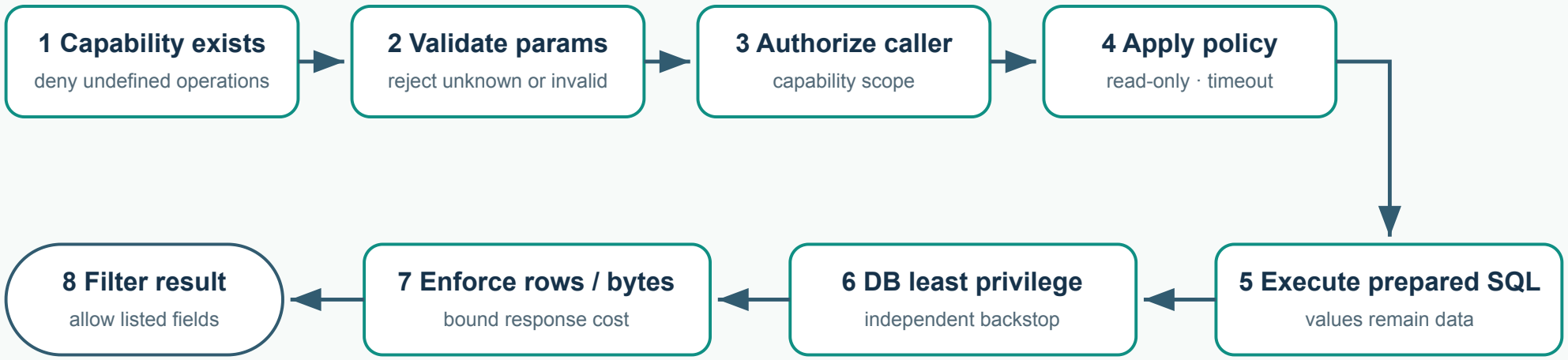
“Mitigated total” counts blocked, constrained, filtered, or detected outcomes. It is not a security score. The full configuration allowed S1 and S17 as intended and allowed the unresolved S18 semantic-content case.

6. Evaluation Results: Control-Layer Effectiveness

Layer	Primary scenarios	Observed effect	Main residual risk
Input contract	S2–S8, S10	Rejected unknown, invalid, nonsensical, oversized, and excessive values	Valid but unauthorized values; cross-field business rules
Execution constraints	S6, S8–S11, S16	Bound values, time, rows, bytes, and writes; kept injected text out of SQL structure	Broad valid reads, repeated calls, driver/engine behavior
Output contract	S11–S15	Removed unexpected fields and detected missing required output	Wrong rows; sensitive meaning in allowed fields
All three	S2–S16 except normal paths	Controlled every adverse structural, runtime, and field-shape case in the simulation	S18 semantic leakage and controls outside the model

The layer counts overlap. That overlap is deliberate: two controls can constrain the same scenario at different stages and preserve a backstop when one fails.

6. Evaluation Results: Defense in Depth



Independent checks limit damage when one control is absent, misconfigured, or bypassed.

The three contracts become materially stronger when capability existence, caller authorization, prepared SQL, and database privilege surround them. MCP exposes `inputSchema` and optional `outputSchema`, but client-visible tool declarations do not replace server enforcement [10]. Tool annotations are hints and must not be trusted as policy.

6. Evaluation Results: Independent Backstops

Scenario	Primary control	Secondary control	Residual control needed
Excessive requested limit	`params.maximum`	`policy.max_rows` and `max_bytes`	Rate/concurrency limits; workload monitoring
Injection-like parameter	Prepared statement	Enum/pattern validation	Avoid dynamic identifiers; least-privilege DB user
Accidental `SELECT *`	Explicit SQL projection	`result` allow-list	Review SQL and schema diffs
Attempted write	`policy.readonly`	Database read-only account	Capability authorization; host security
Undefined capability	Capability allow-list	Scoped discovery and authorization	Agent-side enforcement
Required field removed	`result` mismatch failure	Startup SQL validation	Deployment test against target schema

Duplicate limits are not wasted when they observe different states. Input bounds constrain caller intent, while execution bounds constrain actual work and output.

6. Evaluation Results: Residual Risk

Risk	`params`	`policy`	`result`	Additional control
Caller lacks rights to a valid record	No	No	No	Identity, capability auth, row ownership/RLS
Sensitive text in allowed field	No	No	No	Semantic classification, redaction, DLP
Broad but read-only SQL	No	Partly	Fields only	SQL review, views, row security, DB grants
Repeated requests below limits	No	Partly	No	Rate/concurrency quotas and monitoring
Agent host compromise	No	No	No	Host hardening, secret isolation, incident response
Unsafe contract change	No	No	No	Version control, review, lint, tests, signed release
Engine-specific read-only gap	No	Partly	No	Restricted DB account and engine-specific tests
Wrong global defaults	No	Partly	Partly	Conservative defaults and configuration tests

The contracts bound one capability call. They do not form a complete identity, data-governance, host-security, or availability system.

7. Discussion (1/2)

The experiment supports a defense-in-depth conclusion rather than a claim of security. Input contracts reduce accepted request space. Policy bounds runtime behavior. Result contracts limit named fields that leave the agent. Each layer addresses cases the others allow.

Static definitions improve repeatability, startup validation, change review, and audit interpretation. They also reduce flexibility. The design suits deterministic operational reads such as order status or bounded customer lookup. It is less suitable for exploratory analysis, broad joins, or dynamically generated queries.

Prepared statements address value injection, but a legitimate broad query can still be unsafe. Read-only policy reduces destructive consequences, while confidentiality and availability risks remain. Output allow-lists withstand accidental schema expansion but cannot supply row authorization or interpret free-text sensitivity.

7. Discussion (2/2)

The main failure mode moves from uncontrolled model output to configuration quality. A capability called `export_all_customers` with broad parameters, a long timeout, very high limits, and sensitive result fields can satisfy the schema while violating the security objective. An overly permissive global default can repeat that error across every capability.

Capability configuration should therefore receive the same treatment as security-sensitive code: ownership, code review, version control, structural linting, query validation, scenario tests, and controlled deployment. Human review alone is insufficient because reviewers can miss a default, a new column, or an interaction between SQL and result fields.

The on-premise enforcement implementation also belongs to the trusted computing base. Bugs or a compromised host can bypass normal checks. Database privilege, network controls, secret management, monitoring, and incident response remain necessary.

8. Practical Architecture Guidance (1/2)

Use deny-by-default exposure and define narrow business operations rather than generic table access. Make every accepted parameter explicit, reject unknown names, and prefer enums when the domain is finite. Apply meaningful lower and upper numeric bounds plus string length, pattern, and format constraints.

Keep user values in prepared-statement parameters. Avoid dynamic SQL identifiers where possible. Start read-oriented capabilities with `readonly: true` and a database account that has only the required `SELECT` permissions. Apply explicit timeouts and both row and byte limits. Add concurrency and rate controls outside the capability when repeated calls could exhaust resources.

Generate OpenAPI or MCP discovery only for exposed and authorized capabilities. OpenAPI Schema Objects can describe both input and output data types [7], while server-side enforcement remains authoritative.

8. Practical Architecture Guidance (2/2)

Define an explicit `result` allow-list and avoid `SELECT *`. Fail closed when any required result field is missing. Treat every newly added database column as non-public until a reviewed contract change allows it. Test SQL projection broadening and schema-change scenarios before deployment.

Keep definitions version-controlled. Lint structure, reject unknown configuration fields, validate placeholders, and perform database-aware startup checks where practical. Review global defaults because one unsafe value affects many capabilities.

Log capability name, caller/key identifier, status, error category, latency, and result count where appropriate. Keep SQL, database credentials, authorization headers, raw parameter values, and business payloads out of centralized logs. Monitor error classes and limit hits so operators can distinguish invalid calls, policy enforcement, schema mismatch, and infrastructure failure.

8. Example A: Strongly Constrained Capability

```yaml
capabilities:
  search_customers:
    description: "Search customers by supported status"
    sql: |
      SELECT id, name, email
      FROM customers
      WHERE status = :status
      ORDER BY id
      LIMIT :limit
    params:
      status:
        type: string
        required: true
        enum: ["active", "inactive"]
        maxLength: 8
      limit:
        type: integer
        default: 50
        minimum: 1
        maximum: 500
    policy:
      readonly: true
      timeout: 5s
      max_rows: 500
      max_bytes: 256KB
      expose_in_openapi: true
    result:
      id: { type: integer }
      name: { type: string }
      email: { type: string }
```

The SQL projection, runtime bounds, and egress allow-list are independent controls.

8. Example B: Intentionally Weak Anti-Pattern

```yaml
capabilities:
  query_customers:
    description: "Flexible customer query"
    sql: |
      SELECT *
      FROM customers
      WHERE status LIKE :search
    params:
      search:
        type: string
    policy:
      readonly: false
      expose_in_openapi: true
    # no timeout, max_rows, max_bytes, or result allow-list
```

This anti-pattern accepts an unconstrained string, uses broad SQL, enables write-capable policy without justification, and omits execution and output bounds. It is not recommended for production.

Even adding a schema around this broad operation would not make the business capability narrow. Contract quality matters more than the presence of contract keys.

9. Limitations and Future Work (1/2)

This illustrative simulation does not replace a security audit, penetration test, formal threat model, or production performance test. It assumes the enforcement implementation is correct. It does not cover every database engine, SQL dialect, driver, ORM, API gateway, MCP client, agent framework, transaction semantic, cancellation path, or deployment topology.

Static input contracts cannot express every business rule. Regex and format validation do not establish semantic correctness. Output contracts do not provide row-level authorization and cannot automatically detect sensitive meaning inside allowed free text. Read-only controls leave confidentiality and availability risk. Time, row, and byte limits do not fully prevent resource exhaustion, especially across concurrent or repeated requests.

Database read-only behavior differs by engine and configuration. Workload-specific performance controls remain necessary.

9. Limitations and Future Work (2/2)

The paper focuses on read-oriented capabilities. Writes require caller authorization, approval policy, idempotency, transaction boundaries, concurrency control, rollback behavior, outcome confirmation, and change auditing. Complex workflows spanning several transactions need orchestration and recovery contracts beyond `params`, `policy`, and `result`.

Future work should test multiple real database engines and drivers, property-based and fuzz testing of parameters, schema-evolution matrices, capability-diff review, SQL static analysis, policy verification, and host-compromise assumptions. Further studies should cover write-operation contracts, row-level security, semantic output classification, rate and concurrency limits, cancellation guarantees, and end-to-end authorization.

The local model and generated results remain in `src.zip` so reviewers can reproduce the stated count and change assumptions explicitly.

10. References (1/2)

1. OWASP Cheat Sheet Series, “Input Validation Cheat Sheet.” https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html
2. OWASP Cheat Sheet Series, “SQL Injection Prevention Cheat Sheet.” https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
3. OWASP API Security Top 10, “API4:2023 Unrestricted Resource Consumption.” https://owasp.org/API-Security/editions/2023/en/0xa4-unrestricted-resource-consumption/
4. OWASP Cheat Sheet Series, “LLM Prompt Injection Prevention Cheat Sheet.” https://cheatsheetseries.owasp.org/cheatsheets/LLM_Prompt_Injection_Prevention_Cheat_Sheet.html
5. NIST SP 800-207, “Zero Trust Architecture,” August 2020. https://csrc.nist.gov/pubs/sp/800/207/final
6. JSON Schema, “Validation: A Vocabulary for Structural Validation of JSON,” Draft 2020-12. https://json-schema.org/draft/2020-12/json-schema-validation

10. References (2/2)

7. OpenAPI Initiative, “OpenAPI Specification v3.1.2,” September 19, 2025. https://spec.openapis.org/oas/v3.1.2.html
8. PostgreSQL Global Development Group, “SET TRANSACTION.” https://www.postgresql.org/docs/current/sql-set-transaction.html
9. PostgreSQL Global Development Group, “Client Connection Defaults,” including `statement_timeout`. https://www.postgresql.org/docs/current/runtime-config-client.html
10. Model Context Protocol, “Tools,” specification version 2025-11-25. https://modelcontextprotocol.io/specification/2025-11-25/server/tools
11. Model Context Protocol, “Authorization,” specification version 2025-11-25. https://modelcontextprotocol.io/specification/2025-11-25/basic/authorization

Sources were verified on September 7, 2026. Product-specific behavior and assumptions follow the repository's `architecture.md`; the simulation tests the paper's stated reference model rather than certifying the production implementation.

In Closing

The full input, execution, and output contract produced the best result among the seven simulated configurations. The preferred deployment adds capability-scoped authorization and a database account with minimum read privilege.

This supports Onprest's central architectural direction: expose narrow, deterministic capabilities instead of a database or raw query surface. The conclusion remains conditional on narrow capability design, conservative defaults, correct agent-side enforcement, reviewed changes, and controls for identity, rows, semantics, availability, and host security.

Published by: ViewLegacy LLC.
Author: Asahi Masuda
Publication date: September 7, 2026