American Impact Review

Original Research

Controlling PII Disclosure in Database MCP Connectors

Yevhen Piotrovskyi*

Independent Researcher, Distributed Systems & Software Architecture

ORCID: 0009-0009-2406-5566

* Corresponding author: yevhen.piotrovskyi@gmail.com

OPEN ACCESS

Citation:
Yevhen Piotrovskyi (2026). Controlling PII Disclosure in Database MCP Connectors. *Am. Impact Rev.* 10.66308/air.e2026067

Received: July 15, 2026
Accepted: August 5, 2026
Published: August 10, 2026

DOI:
10.66308/air.e2026067

ISSN: 3071-124X

Abstract

Database connectors for the Model Context Protocol (MCP) let large language model (LLM) agents run SQL against production data, and they return rows verbatim into the model's context window with no filtering. This paper contributes a reproducible benchmark and reference-gateway design, evaluated on synthetic data, for what controls at that boundary can and cannot do. Working over a synthetic-PII SQLite database seeded with canary records and a minimal, MCP-style JSON-RPC client/server, we build a six-rung defense ladder, from an unprotected passthrough to a gateway that combines engine read-only, a single-statement SQL guard, a server-authenticated disclosure policy driven by AST source-column lineage, deterministic reversible tokenization, and egress redaction, and measure each rung's marginal effect with a leak oracle that shares no code with the gateway. Two findings are the point of the paper. First, controls that look sufficient are not: an engine-enforced read-only connection blocks writes but leaves over-exposure unchanged, and a naive allow-list keyed on the output column name drops benign over-exposure to zero yet is defeated by nearly half of our 17-attack suite through trivial aliasing and encoding (SELECT hex(ssn) AS x, SELECT password_hash AS x, two bypasses raised in peer review of an earlier draft). Only when disclosure decisions are made on source columns recovered from the SQL AST (resolving aliases, expressions, aggregates, unions, and CTEs) does attacker success on this suite fall to zero while over-exposure stays at zero. Second, masking direct identifiers is not de-identification: on our data, tokenizing names and emails still leaves 99.2% of customers uniquely re-identifiable from three quasi-identifiers, until date-of-birth is generalized (to 12.5%) and then minimized away (to 0%). Alongside, we report an egress PII detector whose held-out F1 is 0.986 but whose recall on adversarially transformed PII is only 0.20, quantifying why value-level detection must be a backstop, not the primary control. We do not evaluate the current MCP specification/SDK, real MCP server implementations, or real LLMs; those, and a faithful reproduction of the disclosed PostgreSQL connector vulnerability, are stated as future work. All numbers are measured on synthetic data; no real personal data is used.

Keywords: Model Context Protocol, LLM agents, PII, data minimization, SQL lineage, prompt injection, privacy-enhancing middleware, k-anonymity

1. Introduction

The Model Context Protocol (MCP), introduced by Anthropic in November 2024 and since moved to the Linux Foundation's Agentic AI Foundation (December 2025), has become a common way to connect LLM agents to tools - including relational databases [20]. A database MCP server is simple: it advertises a query tool, the model writes SQL, the server runs it, and rows return as text the model can reason over. The convenience is also the hazard: rows return *as stored*. If a customers table holds names, emails, social-security and card numbers, those values enter the model's context window - and from there logs, traces, provider-side telemetry,

and any downstream tool the agent can reach.

Three facts motivate studying this boundary:

1. **Connectors have shipped exploitable.** Datadog Security Labs showed that Anthropic's reference PostgreSQL MCP server could be driven past its BEGIN TRANSACTION READ ONLY wrapper with a stacked COMMIT; DROP …, because the node-postgres simple-query path accepts multiple statements; the package was archived (29 May 2025) and deprecated on npm (10 July 2025) yet still saw ~21,000 weekly downloads, with no CVE assigned [1]. Trend Micro demonstrated SQL-injection and stored-prompt-injection classes against an LLM database agent [2]; Akamai found injection across several third-party database MCP servers [3].
2. **Read-only is not confidentiality.** In the General Analysis / Supabase incident, no SQL injection was needed: a stored instruction in a support ticket caused an agent operating under a high-privilege service_role key - which bypasses row-level security by design - to read data and paste it into an attacker-visible thread [4]. This is the *lethal trifecta*: private data, untrusted content, and an exfiltration channel in one agent [5]. Supabase's own response wraps SQL results with instructions telling the model not to obey embedded commands [21] - a prompt-level mitigation weaker than deciding disclosure on the data itself.
3. **Agents over-fetch at machine speed.** An audit of 6,675 agent tools found 65.42% of transmitted fields redundant or sensitive [6]; a reflexive SELECT * produces data-minimization violations (GDPR Art. 5(1)(c)) at a rate humans do not, and can enable re-identification by correlating quasi-identifiers across otherwise-compliant tables [22].

Despite a fast-growing MCP-security literature - taxonomies [7], large-scale server scans [8], tool-poisoning benchmarks [9], design patterns against prompt injection [10, 11], and practitioner catalogs such as the OWASP MCP/LLM/Agentic Top-10s [19] - we found no peer-reviewed, query-time measurement of PII disclosure at the database-connector boundary. The nearest works analyze server *source code* statically [12], benchmark text-to-SQL leakage without MCP [13], or measure inter-agent channel leakage rather than the database boundary [14]. Two commercial/open tools are closely related and we compare against their *strategy* directly: Lasso's MCP Gateway performs Presidio-based response redaction, and Sanitized DB MCP performs AST-level SQL rewriting with column allow-listing (§7).

What this paper is, and is not

Its contribution is a *reproducible benchmark and reference design*, evaluated on synthetic data, plus a set of **measured, scoped** findings about disclosure controls at a database boundary reached over an MCP-style JSON-RPC channel. It is **not** an empirical evaluation of the current MCP specification, production MCP servers, or LLMs (§7, §9). Concretely:

- **A defense-ladder benchmark** (§6.2) that isolates the marginal effect of each control and shows where intuitive controls fail - including two bypasses raised in peer review, reproduced and then closed by source-column lineage.
- **A gateway design** (§4) whose disclosure decisions are made on AST source columns under a server-authenticated policy, with injective reversible tokenization and corrected collision analysis.

- **Honest ceilings** (§6.1, §6.7, §7): an egress detector that misses 80% of adversarially transformed PII, and a demonstration that tokenizing direct identifiers leaves most records re-identifiable until quasi-identifiers are generalized/minimized.

We are explicit throughout about scope and threats to validity (§7).

2. Background and Threat Model

2.1 The MCP database boundary

An MCP deployment has a **host** (the LLM app), a **client**, and a **server** (the connector holding the database credential). The security-relevant invariant: **the model chooses the query, the connector holds the privilege**, and whatever the connector returns is trusted by the host and enters the context window. By default nothing sits between the raw row and the model.

2.2. Threat model and the worst-case agent

We consider (i) a **malicious query author** (SQL injection), (ii) a **stored-content attacker** who plants text a later agent reads and obeys (the confused-deputy path [4]), and (iii) an **over-privileged but benign agent** that over-fetches. We assume a **fully compliant agent**: it executes whatever query an injection induces. This is conservative for the defender - we never rely on the model refusing - and it means our attacker-success figures are properties of the *connector layer*, not of any particular model. Attacker success is **MUTATE** (a write takes effect) or **EXFIL** (a sensitive value reaches the caller, detected via canary markers or source-column lineage; §5.4).

Formally, a leak needs private-data access A , untrusted content U , and an exfiltration channel $X : leak \Leftrightarrow AUX$ [5]. A gateway cannot remove U (intrinsic to agents) or A (the task needs *some* data); it constrains X so that what crosses the boundary is minimized to an authenticated role's scope and tokenized. Crucially, this bounds - but does not eliminate - disclosure: an agent granted a broad role can still be induced to dump data it is *authorized* to see (§6.8). Least privilege is therefore the primary control; the gateway enforces it and reduces blast radius.

3. The Defense Ladder

We evaluate six rungs, each adding exactly one control, so an outcome is attributable to that control rather than to an incidental change in connection mode. One configurable connector implements all rungs.

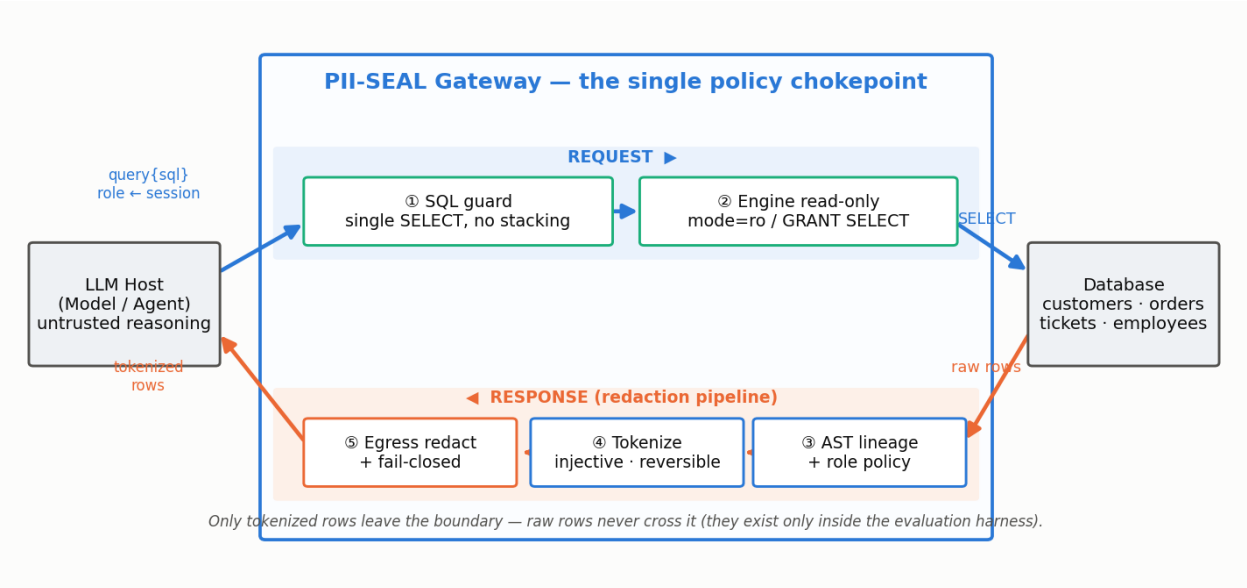
Rung	Adds	Intended effect
L0 raw	nothing (forwards stacked statements, returns rows verbatim)	models today's unprotected connector
L1 +engine-RO	PRAGMA query_only (engine-enforced read-only)	blocks writes
L2 +SQL guard	reject stacked / non-SELECT statements	blocks multi-statement injection
L3 +name allow-list	mask output columns whose name is sensitive	naive column masking
L4 +AST lineage	decide on source columns recovered from the AST	closes alias/encoding/CTE bypasses
L5 full	+ egress value/free-text redaction (backstop)	defense-in-depth for opaque queries

Modeling note

L0 forwards semicolon-separated statements and returns rows verbatim - the behaviour that made the archived PostgreSQL reference server exploitable [1]. SQLite is not PostgreSQL; we model the *stacked-statement class* and cite Datadog for the real execution path. A faithful node-postgres reproduction is future work (§7).

4. Gateway Design

PII-SEAL (rung L5) is middleware at the MCP chokepoint. Every request and response passes through it, so policy is enforced in one auditable place. It is a **reference design evaluated in simulation**, not a hardened product.



Gateway architecture. The PII-SEAL gateway is a single request/response chokepoint. REQUEST (top): the host's tools/call query{sql} - with the role bound to the authenticated session - passes the SQL guard and engine read-only, then hits the database. RESPONSE (bottom): raw rows return through AST source-column lineage + role policy, tokenization, and egress/fail-closed redaction, so only tokenized rows reach the model. Raw rows never cross the boundary.

4.1. SQL analysis and source-column lineage (why the naive allow-list fails)

The guard parses the request (via sqlglot), rejects any request with more than one statement and any statement whose leading verb is not SELECT/WITH. But guarding structure is not enough to control *disclosure*: a single valid SELECT can still exfiltrate. An earlier draft masked columns by their **output name**; peer review demonstrated two one-line bypasses:

SELECT hex(ssn) AS x FROM customers WHERE is_canary=1; - encodes PII
SELECT password_hash AS x FROM employees; - aliases a credential

Both present as a column named x, so a name-keyed allow-list reveals them (whereas a query that names the sensitive column *directly* - SELECT ssn …, or a subquery that projects ssn under its own name - is still caught by the name-list; §6.2). We therefore compute **source-column lineage**: each output column is traced through aliases, function calls, aggregates, set operations, and CTEs/derived tables back to its base (table, column) origins. hex(ssn) AS x resolves to customers.ssn; the CTE WITH t AS (SELECT ssn AS s …) SELECT s FROM t resolves to customers.ssn. Disclosure is decided on those origins, not the alias.

Two lineage subtleties matter for correctness. Value-**reducing** aggregates (COUNT, SUM, AVG) yield a statistic, not a raw value, so COUNT(DISTINCT email) is treated as non-sensitive; value-**preserving** aggregates (MIN, MAX, GROUP_CONCAT) can surface a raw value, so MIN(ssn) is treated as sensitive. When lineage cannot be resolved (opaque expression, a parser failure), the column is marked *incomplete* and the gateway **fails closed** - it redacts every cell rather than trusting the weak egress detector (§4.4). We verify this on an explicitly-unresolvable query: with lineage forced incomplete, the oracle finds no sensitive value in the output.

4.2. Tokenization (corrected collision analysis)

The tokenizer is a keyed, deterministic pseudonymizer $t = T_k(v, \tau)$ built on HMAC-SHA-256, with three properties enforced *by construction*:

- **Determinism:** $v_1 = v_2 \Rightarrow T_k(v_1, \tau) = T_k(v_2, \tau)$, so GROUP BY/COUNT(DISTINCT)/joins on a tokenized key return the same answer.
- **Injectivity (reversibility):** a vault records $t(\tau, v)$ and $(\tau, v)t$; on a token collision with a *different* source value the input is re-salted and re-hashed until unique. Reversibility (write_back, which substitutes originals for tokens an agent echoes) is implemented and tested.
- **Format shape:** tokens keep their type's shape (an SSN token is ddd-dd-dddd, a card token is 16 Luhn-valid digits).

This is **not** format-preserving *encryption*; it is keyed pseudonymization with a vault. A production system should use a real FPE construction (NIST FF3-1) and a KMS-backed vault; our vault is in-memory. Token spaces are **bounded by format**, not by a uniform 48-bit field, so collision risk is per-type. For m distinct values in a space of size S , the birthday estimate is $E[collisions] \approx m^2/2S$:

Type	**Space** S	m **(this corpus)**	$E[collisions]$	**Realized (with resolution)**
EMAIL	2^{48}	4,913	4.3×10^{-8}	0
SSN	10^9	5,000	1.25×10^{-2}	0

Date-of-birth, when tokenized by the gateway, is mapped to a format-preserving pseudonym in a year range **disjoint from the real corpus** (1900 - 1954 vs. the data's 1955 - 2002), so a DOB token neither collides with a real DOB nor discloses the true birth year. Year *generalization* (disclosing the real year) is a distinct minimization option - appropriate only for a role authorized to see coarse dates - and is analyzed separately in §6.7.

4.3. Server-authenticated disclosure policy

Disclosure is governed by an authenticated **role bound to the session**, not by anything the caller sends. The MCP tool exposes only a sql argument; the role is fixed at connect time from the authenticated principal (§5.2). A caller that smuggles "role": "service_role" into the tool arguments is ignored - we verify this with a **privilege-escalation test**: from a session authenticated as analytics, an injected service_role argument leaks **0** canary cells (§6.8). The registry maps a role to the (table, column) pairs it may see in cleartext; a hard deny-list (ssn, card_number, password_hash) is never revealable. We use analytics (aggregates only, no row-level PII), support (contact fields only), and an over-privileged service_role (models Supabase's key) to show that protection is bounded by the granted scope (§6.8).

4.4. Egress redaction and fail-closed (backstop only)

Where lineage is *incomplete*, the gateway **fails closed**: it cannot prove any column safe, so it redacts every cell rather than trusting value detection. For designated free-text columns it additionally scans returned values with a Presidio-style detector (§5.1) and redacts matches. This is deliberately a backstop: §6.1 shows value-level detection is easily evaded, so it must not be the primary control.

5. Methodology

Everything is deterministic given one master seed. **No real personal data is used.**

5.1 Corpus, canaries, detector

A synthetic SQLite database mimics a small SaaS backend: 5,000 customers (name, email, SSN, phone, card, IP, DOB, address), 400 employees, 20,000 orders, 1,200 support_tickets. Because we synthesize every value we hold an exact ground-truth PII manifest. 18% of customers carry names drawn from populations **outside** the detector's dictionary (the realistic long tail), so person-name recall is genuinely below 1. A 200-customer subset is flagged as **canaries** with an improbable prefix; a canary leaks iff its exact string appears in cleartext, an unambiguous signal independent of detector quality.

The detector is a Presidio-style analyzer [15]: regex recognizers, a Luhn checksum for cards, a name gazetteer, and context weighting; each detection carries a confidence, thresholded at τ . We select τ^* on a **dev split** and report on a disjoint **held-out test split**, and separately probe a **hard set** of adversarially transformed PII.

5.2 An MCP-style JSON-RPC boundary

Queries reach the database over a minimal JSON-RPC 2.0 stdio server in the style of MCP, aligned with the protocol's **2025-06-18** revision (initialize / tools/list / tools/call). It is deliberately labelled *MCP-style / legacy*: it does not use the official MCP SDK and predates the current **2026-07-28** specification, which reworked the lifecycle (e.g. removed the initialize handshake); our results are not a statement about that current wire format. Two properties are enforced at this boundary and matter for the claims: the query tool exposes **only** a sql

argument (the disclosure role is bound to the authenticated session, not the request), and the response carries **only** the gateway-processed rows - raw rows never cross the boundary and exist solely inside the evaluation harness as an oracle. In the confused-deputy scenario (§6.8) a **scripted** agent reads a support ticket via the tool, extracts the SQL embedded in the ticket body (a deterministic stand-in for a coerced LLM), and issues it as a second tool call, so the exfiltrated data actually crosses a serialized tool-call envelope.

5.3 Workloads

The benign workload has ten tasks (aggregates, distinct-counts, targeted lookups, a join, two over-broad SELECT *), each running under a role. The adversarial suite has **17 attacks**: stacked DDL/DML, a PRAGMA query_only=OFF write bypass, stacked/ATTACH exfil, and single-SELECT exfil via direct column, alias, hex, hard-deny credential, MIN, concatenation, substring, CTE, UNION, subquery, and the stored-prompt confused-deputy query.

5.4. Metrics and an independent leak oracle

Leaks are judged by an **independent oracle that shares no code with the gateway** - this is essential, because letting the gateway and the evaluator both trust the same lineage analyzer would be circular (a lineage miss could hide a leak from both). The oracle holds the set of all real cleartext PII values in the corpus and, before matching, **decodes** hex and base64 and reconstructs substring chunks, so transformed exfiltration (hex(ssn), substr(ssn,1,5), …) is still caught; canary chunks are matched against the improbable marker's 5-char windows (no false positives on tokens). Benign over-exposure additionally uses a **hand-labeled** table of which output columns carry PII per task and which the role may reveal - written from the SQL by hand, not from any parser. From this: **over-exposure** = unauthorized-cleartext-PII cells / labeled-PII cells; **attacker success** is MUTATE (an observed write) or EXFIL (any returned cell the oracle judges a leak). Detector quality is precision/recall/F1 and average precision; utility is exact rowset equivalence plus an end-to-end tokenized join; latency is gateway processing time per response (wall-clock); re-identification risk is the fraction of records unique under quasi-identifiers ($k = 1$).

6. Results

All numbers below are measured, not illustrative.

6.1. Detector quality - and its adversarial ceiling

With $\tau^* = 0.35$ chosen on the dev split, the detector reaches **precision 0.996, recall 0.977, F1 0.986** on the held-out test cells; context weighting is load-bearing (F1 drops to **0.906** without it). Per-entity F1 is $\approx$1.0 for structured types; person-name F1 is 0.90, limited by the out-of-vocabulary international names we deliberately injected - a genuine false-negative source, not a circular success. On the **hard set** (hex/base64-encoded, dotted, zero-width-split, and OOV-name PII), recall collapses to **0.20**: value-level detection catches only 2 of 10 adversarial transformations. This is the empirical argument for making egress detection a *backstop* and deciding disclosure on lineage instead (Figure 1).

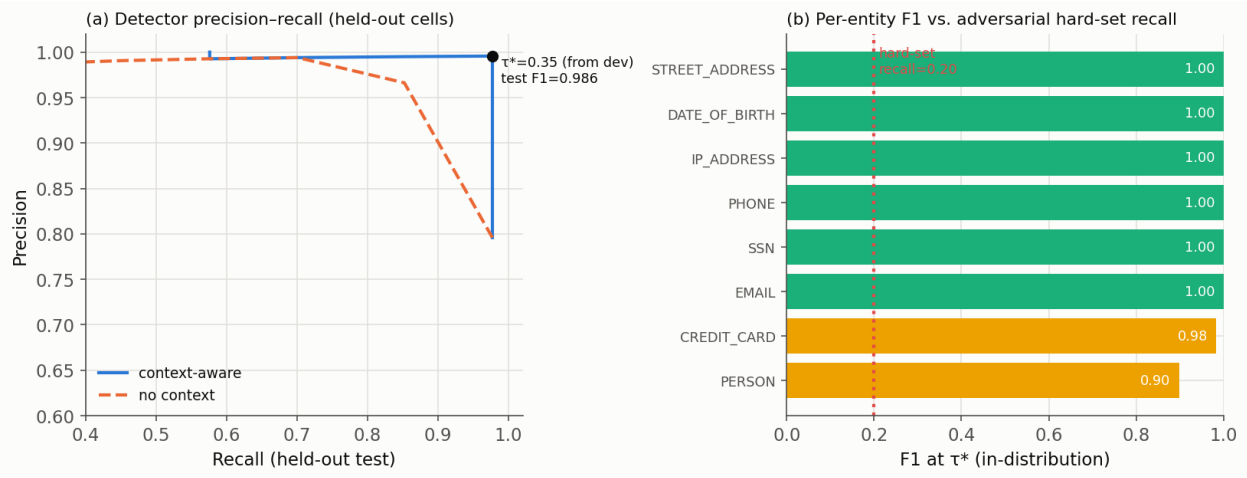


Figure 1. (a) Held-out precision-recall, context vs. no-context, with the dev-selected operating point. (b) Per-entity F1 (in-distribution) against the adversarial hard-set recall of 0.20 (red line).

6.2 The defense ladder (headline)

Figure 2 and Table 1 give each rung's marginal effect on our suite. Engine read-only (L1) blocks all four write attacks - including a PRAGMA query_only=OFF bypass, because we enforce read-only at the VFS level (mode=ro) rather than with the in-band query_only pragma, which we verified an attacker can simply turn off - but it leaves over-exposure at **0.880**: read-only is not confidentiality. The SQL guard (L2) adds stacked-statement rejection. The naive name allow-list (L3) drops benign over-exposure to **0.000**, yet **nearly half the attacks still succeed** (0.471, 8/17): the alias, hex, credential-alias, MIN, concatenation, substring, CTE, and UNION attacks all evade a name-keyed check - while attacks that project a sensitive column under its *own* name (direct SELECT ssn, the subquery variant, the confused-deputy query) are caught by the name-list. Only **AST lineage (L4)** brings attacker success to **0.000** while holding over-exposure at zero; the egress/fail-closed backstop (L5) adds nothing on this fully-lineaged workload but covers opaque cases. The lesson is specific: *structural* guards and *name-level* masking are each insufficient; disclosure must be decided on recovered source columns.

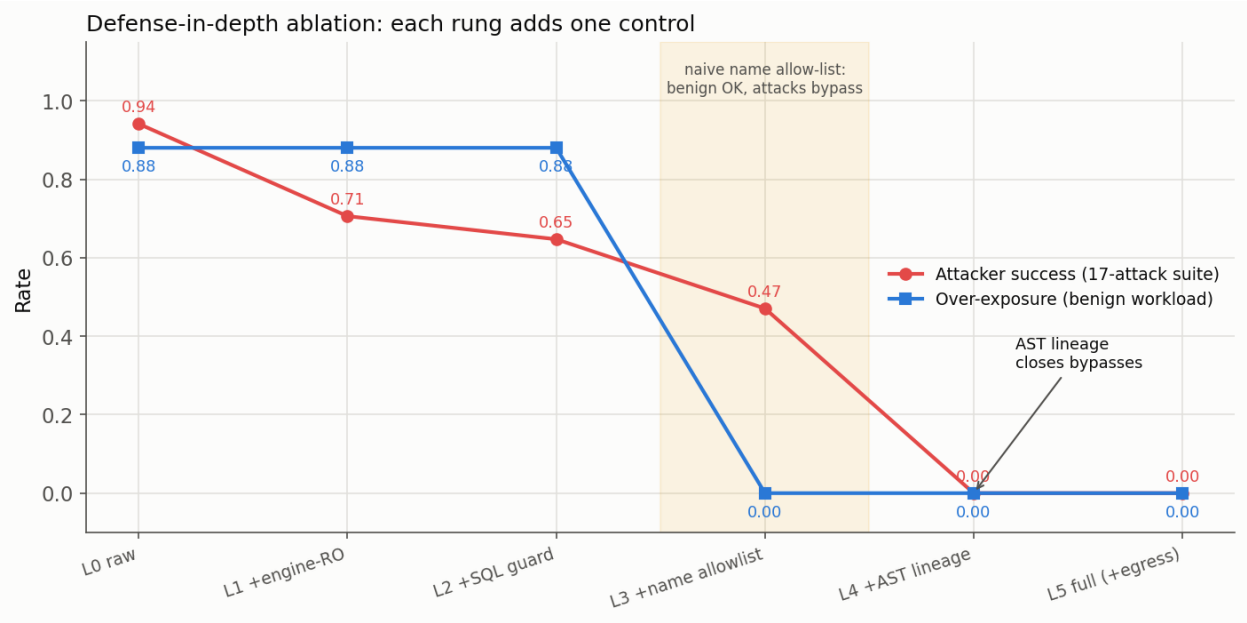


Figure 2. Attacker success (17-attack suite) and benign over-exposure across the six rungs. The naive allow-list (L3) zeroes over-exposure but leaves 47% attacker success; AST lineage (L4) closes the gap.

Table 1. Ablation ladder.

Rung	Attacker success	Over-exposure
L0 raw	0.941	0.880
L1 +engine read-only	0.706	0.880
L2 +SQL guard	0.647	0.880
L3 +name allow-list	0.471	0.000
L4 +AST lineage	0.000	0.000
L5 full (+egress)	0.000	0.000

Table 1 - Ablation ladder.

6.3. Leakage and over-exposure on the benign workload

Grounded in the role policy, the raw connector over-exposes **0.880** of returned sensitive cells; the full gateway reaches **0.000**. Its residual total leakage is exactly the contact fields the support role is authorized to reveal, which both connectors return identically (the targeted lookups in Figure 3b). The two over-broad SELECT * tasks drop from 800/300 cleartext PII cells to 0 (Figure 3b).

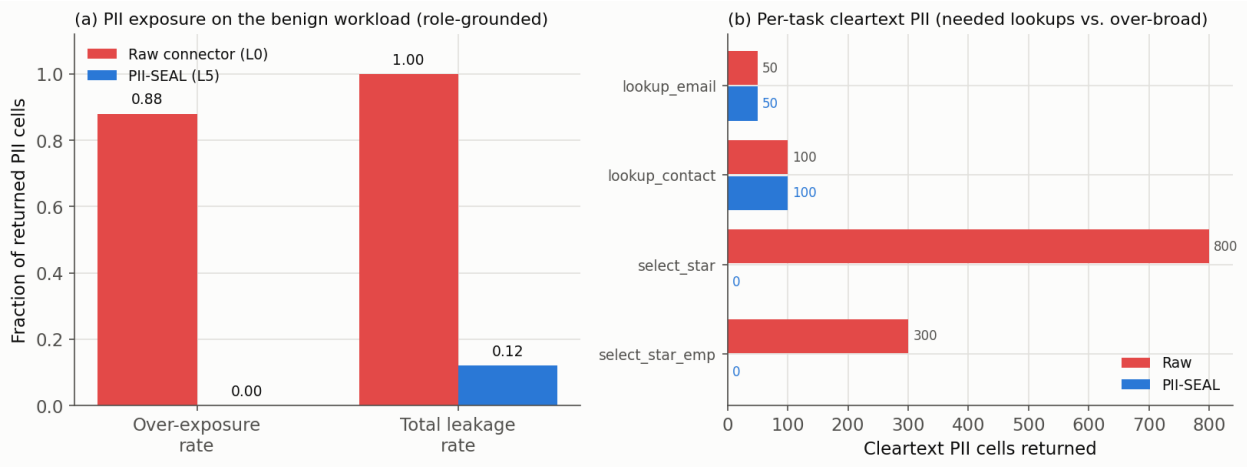


Figure 3. (a) Over-exposure and total-leakage rates, role-grounded. (b) Per-task cleartext PII: authorized lookups are revealed identically; over-broad selects collapse to zero under the gateway.

6.4 Adversarial suite

At L0, **16 of 17** attacks succeed (0.941; only a bare ATTACH, which exfiltrates nothing in our harness, fails), leaking **2,401** canary cells across the suite (the independent oracle counts hex-encoded canaries the earlier lineage-based measure had missed). At L5, **0 of 17** succeed and **0** canary cells leak: six structural attacks are blocked by engine read-only and the guard; the eleven single-SELECT exfil attacks - including the two review bypasses (★ in Figure 4) - are neutralized by lineage-based tokenization/redaction (Table 2).

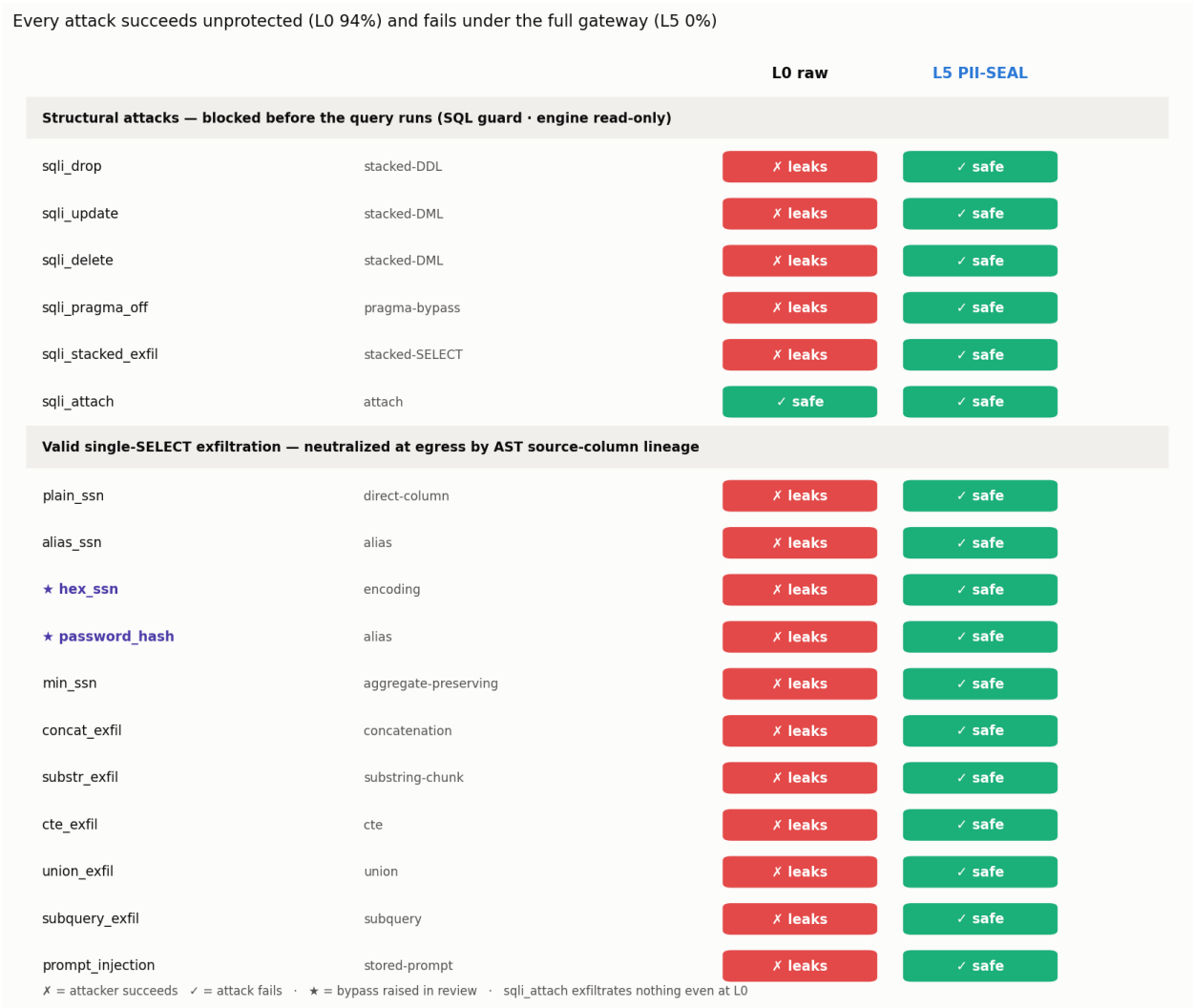


Figure 4. Per-attack outcome, L0 (raw) vs. L5 (full gateway), grouped by the mechanism that stops each attack: structural attacks are blocked before the query runs; valid single-SELECT exfiltration is neutralized at egress by source-column lineage. ★ marks the two bypasses raised in review. (sqli_attach exfiltrates nothing even at L0, so it is "safe" in both columns.)

Table 2. Selected attacks (full list in results/attacks.json).

Attack	Technique	L0	L5	Stopped by
sqli_drop	stacked DDL	×	✓	engine read-only + guard
sqli_pragma_off	PRAGMA query_only=OFF write	×	✓	engine read-only (mode=ro)
hex_ssn ★	hex() encoding	×	✓	AST lineage
password_hash ★	credential via alias	×	✓	AST lineage (hard-deny)
min_ssn	MIN() value-preserving	×	✓	AST lineage
cte_exfil	CTE-hidden source	×	✓	AST lineage (CTE trace)
subquery_exfil	subquery, own column name	×	✓	name allow-list (already at L3)
prompt_injection	stored confused-deputy	×	✓	AST lineage

Table 2 - Selected attacks (full list in results/attacks.json).
(× = attacker succeeds, ✓ = attack fails.)

6.5 Utility preservation

All six aggregate/join tasks return **byte-identical** rows under the gateway (utility 1.00). Directly demonstrating the determinism property: an end-to-end join of orders to customers grouped by the **tokenized** email reproduces the raw grouping exactly - **4,834** groups and identical per-group revenue sums. Tokenizing all 4,913 distinct emails and 5,000 SSNs yields **zero** collisions, consistent with the

$$4.3 \times 10^{-8}$$

/

$$1.25 \times 10^{-2}$$

birthday estimates and the vault's collision resolution.

6.6 Latency

Gateway overhead is linear in returned rows at ≈**18 µs/row** (p50 ≈1.7 ms at 100 rows, 17 ms at 1,000, 87 ms at 5,000 in this frozen run), p99 close to p50 (Figure 5). Latency is the one wall-clock (non-deterministic) result and varies a few µs/row between runs; the figure is drawn from the same frozen latency.json. These are single-threaded Python reference numbers (macOS, Python 3.13); they establish the *shape* (linear, stable), not a production bound. The expensive tail is the 5,000-row SELECT * - the over-broad pattern the gateway exists to discourage.

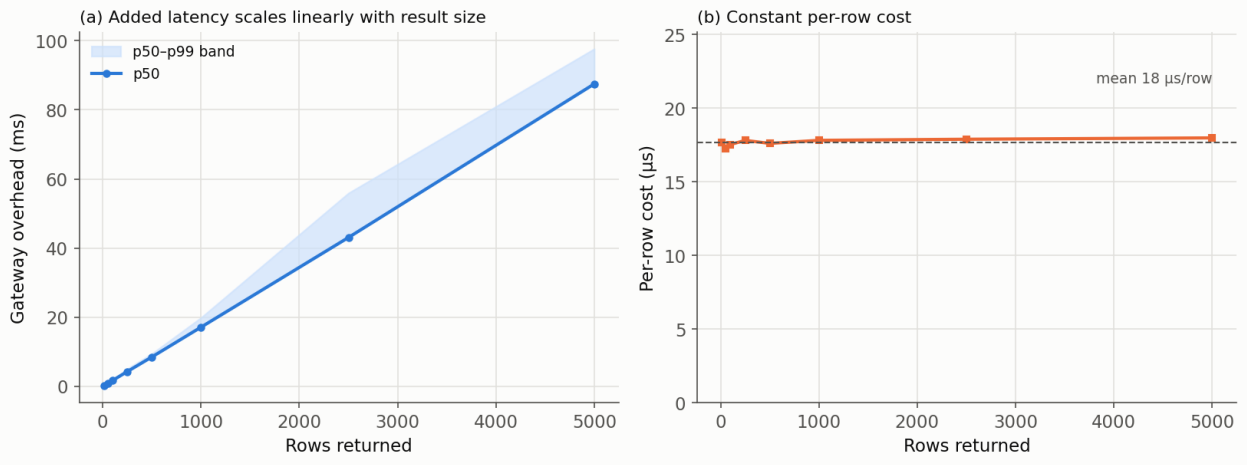


Figure 5. (a) Added latency vs. rows returned (p50 with p50 - p99 band). (b) Constant per-row cost.

6.7. Re-identification and the limit of pseudonymization

Using only three quasi-identifiers - city, region, date of birth - **99.2%** of customers are unique (median $k = 1$). This is why pseudonymizing direct identifiers is not enough: an adversary who tokenizes away names and emails but leaves these fields can still single out almost everyone. Applying DOB → true-year **generalization** (a minimization option a role authorized for coarse dates could invoke) cuts uniqueness to **12.5%** (median $k = 2$); removing DOB entirely reaches **0%** unique (minimum $k = 84$) (Figure 6). Since re-identification probability in a class of size k is at most $1/k$, worst-case odds fall from 1 to ½ to $\leq$1/84. (Note this generalization is a *separate* control from the gateway's default DOB pseudonymization of §4.2, which deliberately discloses neither the value nor the year.)

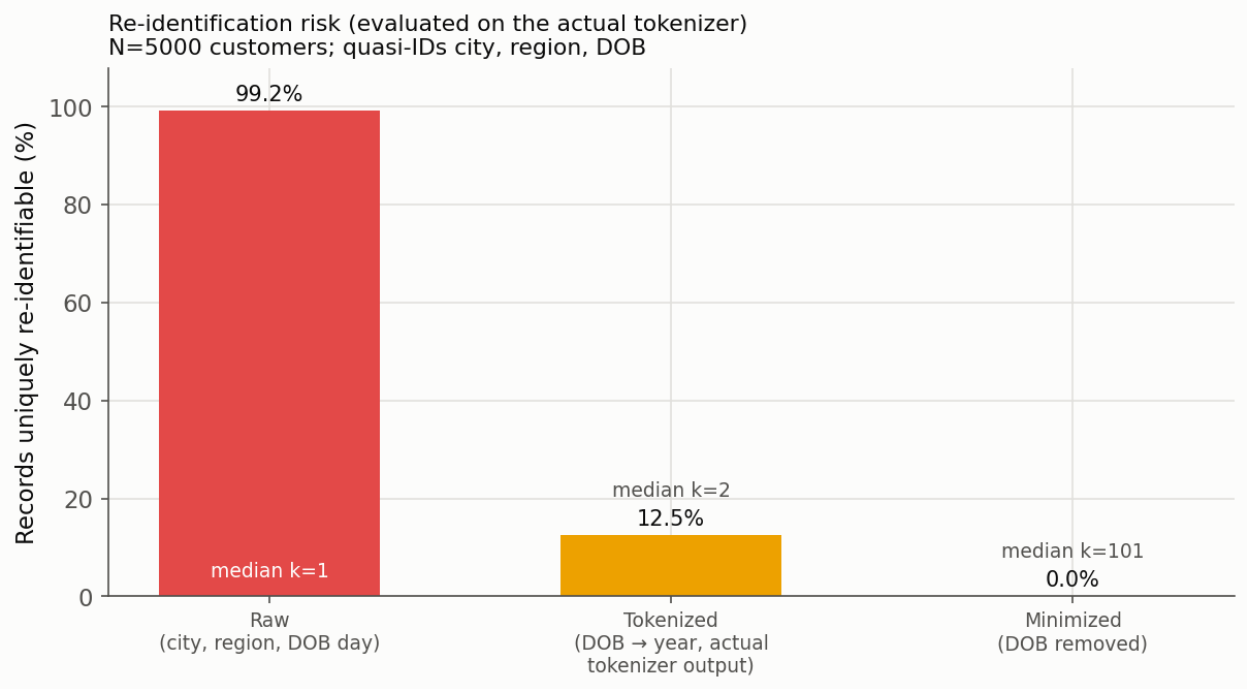


Figure 6. Records uniquely re-identifiable under three treatments of the DOB quasi-identifier: raw, true-year generalization, and removal.

6.8 The confused-deputy scenario, and boundary integrity

The scripted agent reads the injected ticket via a tools/call, extracts the embedded SELECT full_name, ssn, card_number, email FROM customers, and issues it as a second tool call. Against the raw connector it exfiltrates **400** canary cells. Against the full gateway under a session authenticated as least-privilege analytics it exfiltrates **0**. Under a session authenticated as the over-privileged service_role, it still exfiltrates **200** canary emails - the fields that role is *authorized* to see. This is the paper's sharpest caveat: the gateway reduces the blast radius to the authenticated scope but cannot stop misuse of authorized data. Least privilege is the primary control; over-privilege (the Supabase root cause) defeats the rest.

Three boundary-integrity properties are checked directly, since each backs a claim that would otherwise be assertion: (i) **no raw rows on the wire** - the tool response carries only gateway-processed rows (verified by inspecting the serialized payload); (ii) **server-authenticated role** - the tool exposes only a sql argument, and a service_role value smuggled into the arguments from an analytics session leaks **0** canary cells (privilege-escalation test); (iii) **fail-closed** - with lineage forced incomplete, every cell is redacted and the independent oracle finds no sensitive value (§4.4).

6.9 Coverage summary

No single control covers the surface (Figure 7): engine read-only and the guard stop structural attacks; AST lineage closes the alias/encoding/CTE and over-exposure classes; DOB generalization/minimization is the only control addressing quasi-identifier re-identification; egress/fail-closed is a backstop. Removing any column reopens a row.

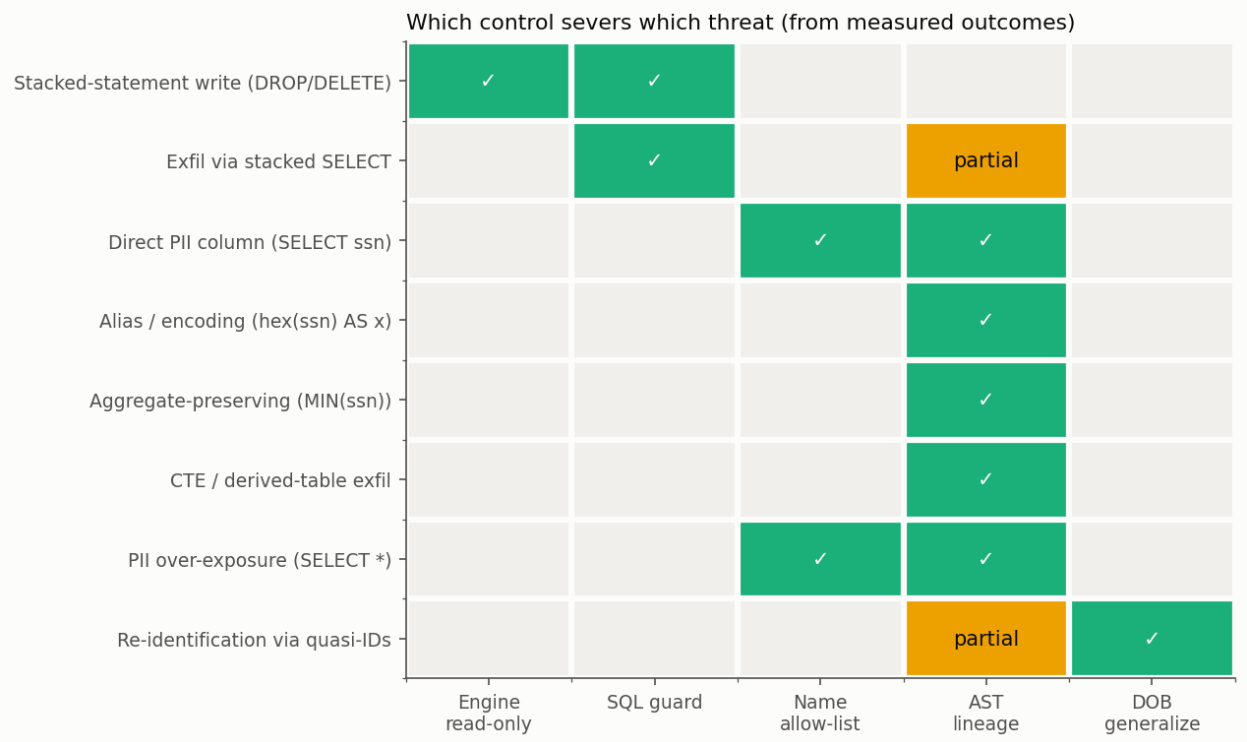


Figure 7. Which control severs which threat, synthesized from the measured ladder outcomes.

7. Scope and Threats to Validity

- **Synthetic data and a scripted agent.** We measure the connector layer under a worst-case compliant agent, but we do **not** run real LLMs. Attacker-success numbers are therefore properties of the connector, and upper bounds on what it permits - not measurements of any model's behaviour.
- **An MCP-*style *boundary, not the current spec/SDK. Our JSON-RPC server is aligned with the MCP 2025-06-18 revision and does not use the official SDK; the current 2026-07-28 specification reworked the lifecycle. We claim a faithful* tool-call boundary* (serialization, session-bound scope, no raw rows on the wire), not conformance to the current wire format.
- **SQLite, not the disclosed PostgreSQL path.** L0 models the stacked-statement *class*; it is not a byte-for-byte reproduction of the node-postgres vulnerability [1], and mode=ro stands in for a Postgres GRANT SELECT role. A faithful PostgreSQL/node-postgres reproduction is future work.
- **Workload-specific magnitudes.** The 0.880 over-exposure and 0.941 attacker success are properties of *this ten-query workload and 17-attack suite*, not estimates for MCP connectors in general. The ladder's *ordering* (which control helps where) is the transferable result; the levels are not.
- **Independent oracle, but bounded coverage.** Leak judgments come from an oracle that shares no code with the gateway and decodes hex/base64/substring chunks (§5.4), so the evaluation is not circular; but an oracle can only catch transformations it decodes - an unusual encoding could evade both gateway and oracle.
- **Detector ceiling and lineage completeness.** Egress detection misses 80% of adversarial transformations (§6.1); the design therefore leans on lineage completeness. Our sqlglot lineage resolves aliases, functions, aggregates, unions, and CTEs on this suite, but adversarial SQL can be made arbitrarily opaque (dynamic SQL, obscure dialect functions); there the gateway fails closed and over-redacts. We do not claim lineage completeness.
- **Bounded, not absolute, protection.** §6.8 shows disclosure is only reduced to the authenticated role's scope; an over-privileged role re-opens leakage of authorized fields.
- **Reference-implementation latency.** Single-threaded Python, wall-clock; establishes shape, not a production bound.

8. Related Work

MCP security

Hou et al. give the first landscape/threat taxonomy [7]; Hasan et al. scan 1,899 servers (7.2% vulnerable, 5.5% tool poisoning) [8]; MCPTox benchmarks tool poisoning [9]; Narajala and Habler translate threats to mitigations [16]; OWASP catalogs the classes [19]. These are ecosystem-level and do not measure PII at query time.

Injection defenses

CaMeL separates control and data flow with capability policies (77% of AgentDojo tasks with provable security vs. 84% undefended) [10]; Beurer-Kellner et al. catalog action-constraining patterns [11]. PII-SEAL is a narrower, complementary boundary enforcement point such policies could delegate egress control to.

Gateways and DB-specific tools (direct comparison)

Our design is not novel in its ingredients. Lasso's MCP Gateway performs Presidio-based response redaction [17] - i.e., our L5 egress backstop, which §6.1 shows is evadable on its own. Sanitized DB MCP performs AST-level SQL rewriting with deny-by-default column allow-listing and type-preserving placeholders [18] - close in spirit to our L4. Our contribution is not the architecture but the **measured comparison**: a ladder that quantifies where name-level vs. source-level control diverges, the two concrete bypasses of name-level control, the re-identification result, and the detector's adversarial ceiling.

The measurement gap

MCPPrivacyDetector statically analyzes 10,655 servers' *source code* (up to 19.1% leakage-risk in Java) [12]; SecureSQL evaluates text-to-SQL leakage without MCP (best model 61.7% vs. 94% human) [13]; AgentRaft measures tool-schema over-exposure (65.42%) [6]; AgentLeak benchmarks inter-agent channels [14]. To our knowledge no peer-reviewed work measures PII disclosure at the database-MCP boundary at query time; this benchmark is a step toward that, not a substitute for evaluating real implementations.

9. Limitations and Future Work

The path to a strong empirical result is clear and is exactly what this study does not yet do: (1) instrument real PostgreSQL and SQLite MCP servers (vulnerable, patched, least-privilege) and capture full MCP traces; (2) run the adversarial suite end-to-end against several real LLMs, separating model-independent connector tests from model-dependent prompt-injection tests; (3) expand the adversarial suite (base64/JSON encodings, nested/dynamic SQL, write-back exfil); (4) replace the in-memory pseudonymizer with FF3-1 FPE and a KMS-backed vault; (5) obtain independent, non-synthetic detector test data. Until then the contribution is a benchmark and a set of scoped findings, not a general claim about MCP connectors.

10. Conclusion

Database MCP connectors put an LLM one SELECT from every record an application holds. In a synthetic benchmark over an MCP-style JSON-RPC channel, we found that the intuitive controls are individually insufficient: engine read-only leaves over-exposure untouched, and a name-level allow-list - which looks perfect on benign traffic - is defeated by half of a small attack suite through aliasing and encoding, including two one-line bypasses. Deciding disclosure on **source columns recovered from the SQL AST** closed those bypasses on our suite while preserving analytic utility exactly; but two honest ceilings remain - value-level egress detection misses most adversarial transformations, and tokenizing direct identifiers leaves records re-identifiable until quasi-identifiers are generalized or minimized - and the gateway's protection is only ever as strong as the authenticated role's least privilege. We hope these findings are reproduced and extended to real MCP servers and models.

References

1 S. Mola, "MCP vulnerability case study: SQL injection in the PostgreSQL MCP server," Datadog Security Labs, 21 Aug 2025. (@modelcontextprotocol/server-postgres archived 29 May 2025, npm-deprecated 10 Jul 2025, ~21k weekly downloads, no CVE; patched fork @zeddotdev/postgres-context-server.) https://securitylabs.datadoghq.com/articles/mcp-vulnerability-case-study-SQL-injection-in-

the-postgresql-mcp-server/

2 Trend Micro Research, "Unveiling AI Agent Vulnerabilities, Part IV: Database Access Vulnerabilities," 2025. Demonstrates SQL-injection and stored-prompt injection against an LLM database agent (Pandora PoC over the Chinook DB). https://www.trendmicro.com/vinfo/us/security/news/vulnerabilities-and-exploits/unveiling-ai-agent-vulnerabilities-part-iv-database-access-vulnerabilities

3 Akamai Security Intelligence Group, "One Fluke, 3-Pattern: MCP Back-End Vulnerabilities," 12 May 2026. https://www.akamai.com/blog/security-research/one-fluke-3-pattern-mcp-back-end-vulnerabilities

4 R. Havaei, R. Liu, M. Li, "Supabase MCP can leak your entire SQL database," General Analysis, 2025 (page updated 2026). Stored injection + service_role (bypasses RLS) exfiltrates via an attacker-visible thread. https://generalanalysis.com/blog/supabase-mcp-blog

5 S. Willison, "The lethal trifecta for AI agents: private data, untrusted content, and the ability to externally communicate," 6 Jul 2025. https://simonwillison.net/2025/Jul/6/supabase-mcp-lethal-trifecta/

6 "AgentRaft: Automated Detection of Data Over-Exposure in LLM Agents," arXiv:2603.07557, 2026 (6,675 tools; 65.42% of fields redundant/sensitive). https://arxiv.org/abs/2603.07557

7 X. Hou, Y. Zhao, S. Wang, H. Wang, "Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions," arXiv:2503.23278, 2025.

8 M. M. Hasan, H. Li, E. Fallahzadeh, G. K. Rajbahadur, B. Adams, A. E. Hassan, "Model Context Protocol (MCP) at First Glance," arXiv:2506.13538, 2025 (1,899 servers; 7.2% vulnerable, 5.5% tool poisoning).

9 "MCPTox: A Benchmark for Tool-Poisoning Attacks on Real-World MCP Servers," arXiv:2508.14925, 2025.

10 E. Debenedetti, I. Shumailov, T. Fan, J. Hayes, N. Carlini, D. Fabian, C. Kern, C. Shi, A. Terzis, F. Tramèr, "Defeating Prompt Injections by Design (CaMeL)," arXiv:2503.18813, 2025 (77% AgentDojo w/ provable security vs. 84%).

11 L. Beurer-Kellner et al., "Design Patterns for Securing LLM Agents against Prompt Injections," arXiv:2506.08837, 2025.

12 B. Yan, M. Xu, Y. Yang, B. Ma, X. Dai, J. Li, Y. Zhang, "'What Happens Locally, Leaks Globally': Detecting Privacy Leakage Risks in MCP Servers," arXiv:2606.21338, 2026 (10,655 servers; up to 19.1% leakage-risk).

13 Y. Song, R. Liu, S. Chen, Q. Ren, Y. Zhang, Y. Yu, "SecureSQL: Evaluating Data Leakage of LLMs as Natural Language Interfaces to Databases," Findings of EMNLP 2024, pp. 5975 - 5990. https://aclanthology.org/2024.findings-emnlp.346/

14 F. El Yagoubi, G. Badu-Marfo, R. Al Mallah, "AgentLeak: A Benchmark for Internal-Channel Privacy Leakage in Multi-Agent LLM Systems," arXiv:2602.11510, 2026.

15 Microsoft, "Presidio: Data Protection and De-identification SDK." Regex recognizers, spaCy NER, context enhancement, checksum validation. (No official precision/recall benchmark is published.) https://microsoft.github.io/presidio/

16 V. S. Narajala, I. Habler, “Enterprise-Grade Security for the Model Context Protocol,” arXiv:2504.08623, 2025.

17 Lasso Security, “MCP Gateway” (open source), 2025. Presidio-based response sanitization / PII masking for MCP. https://github.com/lasso-security/mcp-gateway

18 “Sanitized DB MCP” (open source), 2025. Database MCP server with AST-level SQL rewriting, deny-by-default column allow-listing, and type-preserving placeholders. https://pypi.org/project/sanitized-db-mcp/

19 OWASP, “MCP Top 10” (MCP01 Token Mismanagement, MCP03 Tool Poisoning); “Top 10 for LLM Applications 2025” (LLM01 Prompt Injection, LLM02 Sensitive Information Disclosure); “Top 10 for Agentic Applications 2026.” https://genai.owasp.org/

20 Anthropic, “Introducing the Model Context Protocol,” 25 Nov 2024; MCP Specification, authorization revision 2025-06-18 (OAuth 2.1 resource-server, RFC 8707). MCP donated to the Linux Foundation’s Agentic AI Foundation, 9 Dec 2025. https://modelcontextprotocol.io/specification/2025-06-18

21 Supabase, “Defense in Depth for MCP Servers,” 2025. Wraps SQL results with instructions discouraging the model from obeying embedded commands - a prompt-level mitigation weaker than data-level control. https://supabase.com/blog/defense-in-depth-mcp

22 GDPR Art. 5(1)(c) (data minimization) & Art. 83(5) (fines up to €20M / 4% turnover); CCPA/CPRA Cal. Civ. Code §1798.155; HIPAA/HITECH penalty tiers (inflation-adjusted).