From SQL Generation to Tool Selection: A Domain-Oriented Pattern for MCP Servers

Bartolomeo Bogliolo

August 2026

Abstract

Agents built on Large Language Models (LLMs) increasingly reach enterprise data through the Model Context Protocol (MCP), and many MCP database servers maximize flexibility by exposing a single generic SQL execution tool. This paper proposes the \\textbf{Domain-Oriented Tooling Pattern}: instead of generating SQL at query time, the model selects from a small set of domain-aligned tools whose parameterized queries encapsulate schema navigation, joins and business rules on the server side. We formalize the pattern around three architectural invariants and introduce \\textbf{Model Demotion}, the observation that replacing SQL synthesis with intent classification lowers the model tier required to serve routine requests. As a reference implementation we present \\textbf{MCP Blueprint}, an open-source framework that defines domain tools declaratively as YAML metadata plus parameterized SQL files. A public reproducibility benchmark compares three server designs — raw SQL execution, a thin generic pack, and a verticalized domain pack — on four local models (3B–8B) over seventeen enterprise-style tasks: the verticalized pack scores 0.939 pooled mean accuracy against 0.666 for raw SQL and 0.605 for the generic pack, with every model improving and the smallest gaining most (0.583 → 0.929). All harness code, prompts, gold answers, packs and per-cell results are publicly available.

Code: https://github.com/meob/mcp-blueprint Benchmark artifacts: https://github.com/meob/mcp-blueprint-benchmark

Introduction

Large Language Models are rapidly becoming the primary interface between users and enterprise information systems. The Model Context Protocol (MCP) [1] accelerates this trend by standardizing how AI agents discover and invoke external tools, and database connectivity is among its most common applications: an MCP server fronting a relational database lets agents answer business questions directly from operational data.

When building such a server, developers face a design decision that shapes everything downstream: what should the server actually expose? The fastest path — and a recurring pattern in published connectors — is a single generic tool such as `execute_sql(query)`, optionally accompanied by schema metadata in the system prompt. This choice maximizes short-term flexibility: the agent can explore, join and filter anything the database credentials allow, with zero upfront domain modeling.

It also transfers responsibilities to the model that software engineering traditionally assigns to the application layer: schema discovery, relationship inference, dialect handling, business-rule interpretation, query optimization and result validation. These concerns consume reasoning capacity, in-

troduce non-determinism where determinism is cheapest to guarantee, and widen the deployment's security surface. In practice they push teams toward large, expensive frontier models simply to compensate for an interface that exposes implementation details instead of business concepts.

This paper argues that the abstraction layer — not the language model — is the decisive design variable, and proposes replacing SQL generation with **tool selection**: the server exposes a small set of domain-aligned operations ("verticalized" tools), each backed by pre-authored parameterized SQL that encapsulates joins and business rules.

The contributions of this paper are:

1. We formalize the **Domain-Oriented Tooling Pattern** as three architectural invariants and relate it to established abstraction layers such as object-relational mapping, REST resources and semantic layers (Section 3).
2. We introduce **Model Demotion** as an engineering heuristic: reducing interface complexity converts open-ended SQL synthesis into intent classification and slot filling, lowering the model tier required for routine requests (Section 5).
3. We present **MCP Blueprint** [20], an open-source framework implementing the pattern through declarative YAML/SQL pack definitions decoupled from protocol infrastructure.
4. We release a **public reproducibility benchmark** [21] comparing raw SQL access against two MCP tool-pack designs across four local models and seventeen enterprise-style tasks (Section 6). The verticalized pack dominates every measured dimension — pooled accuracy 0.939 versus 0.666 (raw SQL) and 0.605 (generic thin-tool pack); the smallest 3B model matches every larger configuration; cost per correct answer drops by factors of roughly 2–12×. A superficially similar generic pack underperforms even raw SQL, indicating that tool *design*, not tool *existence*, creates value.

Section 2 reviews background and motivation. Section 3 defines the pattern, Section 4 summarizes the reference implementation, and Section 5 develops Model Demotion. Section 6 reports the benchmark design and results, Section 7 discusses implications, Section 8 surveys related work, and Section 9 concludes.

Background and Motivation

The Model Context Protocol

The Model Context Protocol [1] standardizes how LLM applications discover and invoke external capabilities. An MCP server advertises typed tools — name, description, JSON-Schema parameter contracts — and an MCP client surfaces these schemas to the model, which decides when and how to call them. Transports include stdio for local processes and Streamable HTTP for remote deployments.

Because any capability can be packaged as a tool, MCP servers now front databases, SaaS APIs, filesystems and internal services. For relational data specifically, the fastest way to ship a connector is to wrap a database connection in one generic query tool and hand the schema DDL to the model.

Generic Query Interfaces: A Common Design Choice

A representative generic tool looks like this:

```
{
  "name": "execute_sql",
  "description": "Executes an arbitrary SQL query against the target database.",
  "parameters": { "query": "STRING" }
}
```

The design is genuinely attractive at first: it requires no domain modeling up front, covers the long tail of ad-hoc questions during prototyping, and delegates every difficult decision to a model that is often remarkably capable of improvising SQL.

It also transfers responsibilities that software engineering traditionally assigns to the application layer. Consider a question such as *"Which invoices are still unpaid for customer ACME?"* Answering it through raw SQL access requires the model to:

- discover which tables hold customers, invoices and payments;
- infer how those entities relate and which joins apply;
- find out how "unpaid" is represented in this particular schema;
- decide which business filters must accompany the obvious ones;
- produce dialect-correct, reasonably efficient SQL;
- validate the shape of the returned rows before answering.

None of these steps is the user's question. They are implementation concerns that recur — with fresh probabilistic variation — on every single request.

Operational Limitations of Generic Interfaces

Four consequences follow from placing these burdens on the model.

1. Schema and context overhead. To formulate correct SQL the model must first ingest schema metadata. For enterprise schemas with hundreds of tables and thousands of columns this consumes large portions of the context window, recurs across agent iterations, inflates cost and latency, and dilutes attention on the user's actual task.

2. Probabilistic execution risk. Generated queries are synthesized anew each run and may scan unindexed tables, form Cartesian products through missing join conditions, or fetch far more data than needed (`SELECT *`). In concurrent production environments a single pathological query can starve connection pools or induce lock contention.

3. Business-rule re-derivation. Relational schemas normalize entities and leave business semantics implicit. Whether an account counts as "active", or a rental as "overdue", typically depends on multi-column state evaluations and temporal comparisons encoded in conventions rather than constraints. A model writing raw SQL must re-derive these rules on every invocation, producing interpretations that vary across runs, models and prompt phrasings.

4. Enlarged security surface. A generic execution tool grants broad read capability by construction. Even with read-only credentials it remains exposed to indirect prompt injection [17], bulk data harvesting, and resource-exhaustion patterns such as deep recursive common table expressions. Because nothing in the interface constrains what a query may express, every mitigation must be re-imposed at the query boundary.

These limitations are not a verdict on text-to-SQL research, which continues to advance rapidly [9]–[12]. They reflect deploying an exploration-grade interface into a production serving path.

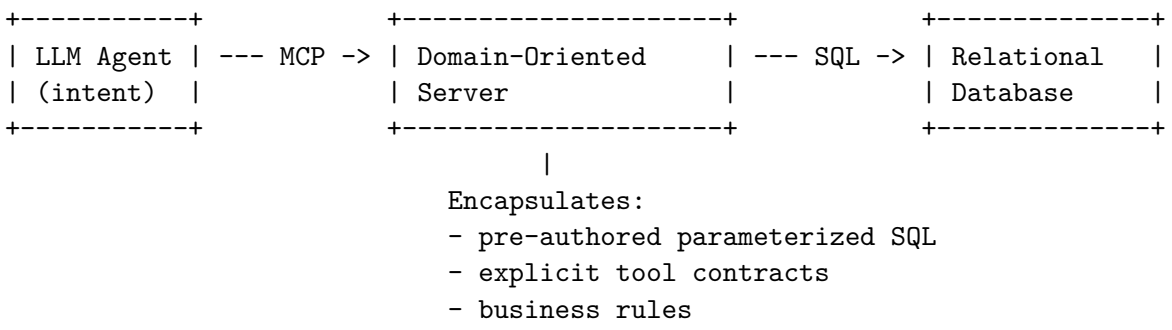
The Domain-Oriented Tooling Pattern

Core Philosophy

The pattern is captured by a single directive:

> **Do not expose the database. Expose the domain.**

Under this paradigm the MCP server acts as a domain gateway. The LLM remains an orchestrator that requests semantic information or triggers domain actions; every data-access decision stays encapsulated inside the server.



Dimension	Generic SQL interface	Domain-oriented interface
Tool granularity	`execute_sql()`	`customer_account_summary()`, `recommend_films()`
Schema knowledge	Discovered from context per request	Implicit in reviewed tool contracts
Joins	Generated per request	Pre-authored and optimized
Business rules	Re-derived by the model	Embedded in server-side SQL
Model tier	Frontier models typical	Small models frequently sufficient
Output shape	Varies per run	Stable, documented columns

The transition mirrors familiar precedents. Object-relational mappers abstracted SQL behind programming-language objects; the move from raw RPC to RESTful resources [3] replaced unconstrained remote invocation with bounded, self-describing operations; domain-driven design [2] supplies the vocabulary: tools correspond to operations of a bounded context, not to storage primitives. The Domain-Oriented Tooling Pattern applies the same discipline to a new class of client — a probabilistic one that benefits even more from narrow, well-documented contracts.

MCP Blueprint: A Reference Implementation

MCP Blueprint [20] operationalizes the pattern without bespoke boilerplate for each domain. It replaces imperative server code with declarative configuration files ("packs") and enforces a strict separation of concerns:

- **Engine layer** — protocol serialization, stdio and Streamable HTTP transports, connection pooling across database engines, parameter validation, response caching and error handling; domain-agnostic and shared by all packs.
- **Domain pack layer** — YAML tool definitions, external parameterized SQL files and pack metadata; self-contained artifacts that can be versioned, reviewed, tested and ported across database engines.

```
mcp-blueprint/
  blueprint/            # Core Python engine (transports, pooling, validation)
  config/               # Server configuration (database URI, engine selection)
  packs/
      sakila/           # Sakila Domain Pack
          pack.yaml     # Engine compatibility & pack declaration
          tools/        # Declarative YAML tool definitions
          sql/          # External parameterized SQL files
```

Adding a domain operation is a configuration change rather than a code change: an author writes one YAML definition and one reviewed SQL file and commits both. Appendix A shows a complete example from the verticalized pack used in Section 6, including how an availability rule and an optional parameter are handled directly in SQL.

Model Demotion

Lowering the Cognitive Threshold

A central consequence of the pattern is what we call **Model Demotion**: reducing interface complexity lowers the model tier required to serve routine requests.

Through a generic SQL interface, answering a business question requires open-ended synthesis: navigate the schema, choose join paths, translate intent into dialect-correct SQL, execute it, inspect errors and recover. Through a domain-oriented interface, the same request reduces to intent classification and slot filling — recognizing which operation applies and extracting a few typed parameters.

```
+-------------------------------------------------------------------+
|                       GENERIC QUERY INTERFACE                     |
|   User intent -> frontier LLM -> dynamic SQL generation -> DBMS   |
+-------------------------------------------------------------------+

+-------------------------------------------------------------------+
|                     DOMAIN-ORIENTED PATTERN                       |
|   User intent -> SLM (intent/slots) -> pre-authored SQL  -> DBMS  |
+-------------------------------------------------------------------+
```

The distinction matters economically and operationally:

- **Cost and latency** — classifying intent over a handful of tools is an easier decoding problem than multi-statement code generation, so it can be served by small local models or inexpensive API tiers.
- **Failure modes** — synthesis failures are open-ended (wrong joins, invalid syntax, hallucinated columns); selection failures are bounded (wrong tool, missing parameter) and therefore easier

to detect, log and repair.
- **Determinism** — with fixed tools, the semantic content of a request lives on the server rather than in model weights, so behavior converges across models at temperature 0.

We present Model Demotion as an engineering heuristic rather than a law: it applies to bounded, recurring retrieval workflows, not to open-ended analytical exploration.

Partitioning Data Access by Risk

Enterprise data requests are highly skewed: a bounded set of core operations covers most routine traffic. Following a **95/5 heuristic**, we expect the large majority of enterprise retrieval scenarios to be expressible as a curated collection of semantic operations, with a long tail of exceptional requests.

The pattern partitions workload accordingly:

1. **The routine majority.** Recurring operational requests are served deterministically by pack tools, routed by lightweight models, without exposing raw tables.
2. **The exceptional remainder.** Requests outside current coverage route to a human-in-the-loop workflow: a domain engineer authors, reviews and commits a new YAML/SQL definition, permanently absorbing that case into the deterministic set.

This creates a continuous improvement cycle — human expertise is encoded once into a pack and then served repeatedly by cost-effective models. Generic SQL access, by contrast, re-pays the cost of schema understanding on every single request.

Section 6 tests the central claim of Model Demotion empirically: whether small local models, given domain-oriented tools, can match or exceed larger configurations working through raw SQL access.

A Public Reproducibility Benchmark

To quantify the effect of interface design we built a standalone benchmark harness and released it publicly [21]. The harness runs identical task prompts against pluggable MCP server configurations, scores every run against gold answers computed live from the database, and records one JSON file per cell (model × approach × task × repetition) containing token counts, latency, agent steps, tool-call traces and per-check results. All components needed to reproduce the study — harness code, task definitions, scoring rules, gold-answer logic, the frozen packs behind approaches B and C, the DDL used by approach A, and a run manifest pinning framework commit and pack versions — are committed to the repository together with the complete per-cell record set of the reported run.

Design and Setup

Approaches. Three MCP server configurations expose the same Sakila sample database (PostgreSQL port):

	Approach	Surface
A	Raw SQL	A single `execute_sql` tool; the DDL of the six task-relevant tables (`customer`, `film`, `category`, `film_category`, `inventory`, `rental`) is embedded in the system prompt. The schema fits entirely in context, so A is not handicapped by prompt size; schema understanding, SQL synthesis and multi-step orchestration are left to the model.
B	Verticalized pack	MCP Blueprint loading `packs/sakila` v0.5.0: five domain tools (`customer_account_summary`, `rental_history`, `recommend_films`, `film_stock`, `search_customer`) that accept human-readable identifiers (names, titles) and encapsulate all joins and business rules in pre-authored SQL.
C	Generic thin-tool pack	MCP Blueprint loading a deliberately shallow pack (`search_customer`, `search_films`, `get_customer_rentals`, `get_film`): table-oriented tools with minimal descriptions — a first-pass surface a developer might ship before verticalizing. C isolates the contribution of tool *design* from tool *existence*.

Models. Four instruction-tuned models served locally through Ollama:

Model	Params	Tier
llama3.2:3b	3B	SLM
qwen2.5:3b	3B	SLM
qwen2.5:7b	7B	Medium
llama3.1:8b	8B	Medium

Model	Params	Tier

Two smaller models were excluded during bring-up because their Ollama builds do not support tool calling (HTTP 400): gemma2:2b and phi3:mini.

Protocol. Temperature 0; seed 42; context window 8192 tokens; at most 10 agent steps; three repetitions per cell. The design is 4 models × 3 approaches × 17 tasks × 3 repetitions = 612 planned cells, of which 609 completed (99.5%); three cells (qwen2.5:3b / `service_case` / approach A) were lost to a stdio pipe hang during MCP server startup. The run was recorded on 2026-08-20 against MCP Blueprint commit `b386d58`.

Scoring. Each task defines a rule-based check set evaluated against gold answers computed live from the database; no LLM judge is involved. Free-form titles are compared by fuzzy matching (SequenceMatcher ratio ≥ 0.72). Workflow checks (tool-call sequences, argument validation) apply only to approaches B and C; approach A is scored solely on its final answer text. A cell's score is passed checks over total checks, in [0, 1].

Task Suite

Seventeen customer-facing tasks stress different capabilities:

Category	Tasks
Customer lookup	`find_customer`, `not_found`
Rental state & standing	`rental_history`, `return_verify`, `service_case`, `overdue_report`, `good_standing_recommend`
Recommendation	`recommend_category`, `recommend_rating`, `g_available`, `avoid_on_loan`, `upsell_seen`, `not_rented`
Catalog details	`film_details`, `store_availability`
Multi-step workflow	`customer_workflow`
Edge cases	`rental_empty`

Negative-filtering tasks (`not_found`, `not_rented`, `avoid_on_loan`, `upsell_seen`) and multi-step workflows are deliberately included because they stress composition and business-rule application rather than single-query lookup. Full prompts appear in Appendix B.

Accuracy Results

Pooled across models, the verticalized pack dominates every accuracy metric, while the generic pack underperforms even raw SQL:

Metric	A (Raw SQL)	B (Verticalized)	C (Generic)
Mean score	0.666	**0.939**	0.605
Fully-correct cells	67/201 (33%)	**174/204 (85%)**	63/204 (31%)
Zero-score cells	7 (3%)	**3 (1%)**	19 (9%)

Approach B holds the accuracy ceiling for every model:

Model	A	B	C
llama3.2:3b	0.583 (6/51)	**0.929 (42/51)**	0.419 (0/51)
qwen2.5:3b	0.684 (17/48)	**0.902 (42/51)**	0.602 (14/51)
qwen2.5:7b	0.647 (20/51)	**0.958 (45/51)**	0.631 (19/51)
llama3.1:8b	0.750 (24/51)	**0.966 (45/51)**	0.769 (30/51)

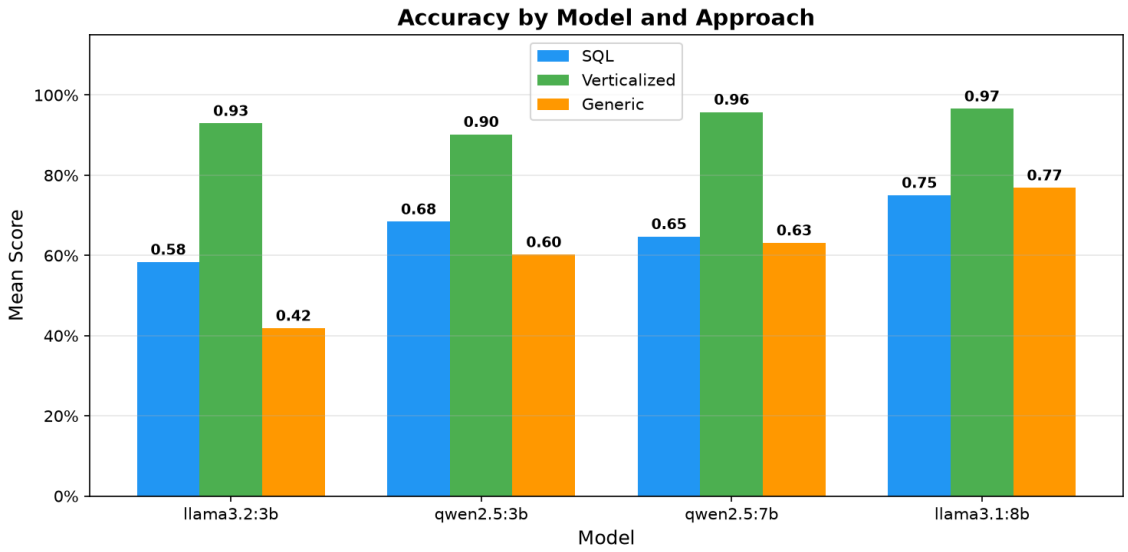


Figure 1: Pooled mean accuracy by model and approach. The verticalized pack (B) leads at every model size, with the largest gain on the smallest model (llama3.2:3b), consistent with the Model Demotion hypothesis.

Three observations stand out. First, B never drops below 0.90 on any model: the benefit does not depend on model scale. Second, the smallest model gains the most — llama3.2:3b improves from 0.583 under raw SQL (with only 6/51 fully-correct cells) to 0.929 with domain tools (42/51), which is the empirical signature of Model Demotion. Third, the generic pack C pools below even raw SQL (0.605 vs 0.666), trailing A on three of four models; exposing tools without designing them can be worse than not exposing tools at all. Section 6.5 analyzes why.

Token and Latency Efficiency

Mean per-cell token cost is comparable across designs; the decisive efficiency metric is cost per *correct* answer, where the designs separate sharply:

Approach	Mean tokens/cell	vs B	Tokens per correct answer	Seconds per correct answer
B (Verticalized)	3,056	—	**3,582**	**5.2**
C (Generic)	2,894	–5.3%	9,372	20.7

Approach	Mean tokens/cell	vs B	Tokens per correct answer	Seconds per correct answer
A (Raw SQL)	3,953	+29.4%	11,858	51.9

Mean latency per cell follows the same ordering: 4.4 s (B), 6.4 s (C) and 17.3 s (A), with mean agent steps of 2.1, 2.4 and 2.4 and mean tool calls of 1.1, 1.7 and 1.6 respectively. B is simultaneously the most accurate and the fastest design in the study.

Cost per correct answer improves for every model when moving from raw SQL to the verticalized pack:

Model	A: tokens/correct	B: tokens/correct	A: s/correct	B: s/correct	Reduction
llama3.2:3b	31,476	**2,723**	49.5 s	**3.4 s**	$ 11.6×$
qwen2.5:3b	17,467	**4,766**	125.1 s	**5.0 s**	$ 3.7×$
qwen2.5:7b	8,464	**4,331**	33.2 s	**5.6 s**	$ 2.0×$
llama3.1:8b	5,808	**2,531**	16.3 s	**6.7 s**	$ 2.3×$

The smallest model is the clearest beneficiary of Model Demotion. Under raw SQL, llama3.2:3b consumed 3,703 tokens and 5.8 seconds per cell while solving few cells completely; under the verticalized pack it consumed 2,242 tokens and 2.8 seconds per cell while solving most of them — reducing tokens per correct answer from 31,476 to 2,723.

Equivalently: the cheapest fully-correct cell in the study is llama3.2:3b with domain tools at roughly 2.7k tokens and 3.4 s, while qwen2.5:7b with raw SQL spends about three times as many tokens and ten times as long per correct answer. Equivalent quality at a fraction of the inference budget is the economic form of the pattern's central claim.

Tool Design versus Tool Existence

The contrast between approaches B and C is the benchmark's sharpest finding. Both are MCP Blueprint packs exposing the same database through parameterized tools — yet B reaches 0.939 while C stalls at 0.605, below even raw SQL. The difference lies entirely in how the tools are designed:

Dimension	B (Verticalized)	C (Generic)
Tool semantics	Domain operations (`customer_account_summary`, `recommend_films`)	Table-oriented (`search_films`, `get_customer_rentals`)
Parameter contracts	Human-readable names; descriptions tuned for matching	Bare IDs or minimal labels
Business logic	Encapsulated server-side (standing flag, overdue detection)	Re-derived by the model

Dimension	B (Verticalized)	C (Generic)
Descriptions	Rich guidance steering correct workflows	Minimal, forcing multi-step composition

A thin tool surface leaves all reasoning with the model while removing its freedom to compensate through arbitrary SQL: it constrains the action space without raising answer quality. The practical implication is that the tool surface itself — clear parameters, explicit contracts, logic pushed server-side — sets the ceiling for answer quality; neither larger models nor the mere presence of a tool layer substitute for it.

Per-Task Analysis

Ordering tasks by the gap between B and A shows where verticalization contributes most:

Task	A	B	C	Δ(B–A)
`recommend_category`	0.500	**1.000**	0.833	+50.0pp
`avoid_on_loan`	0.500	**1.000**	0.667	+50.0pp
`upsell_seen`	0.500	**1.000**	0.750	+50.0pp
`film_details`	0.562	**1.000**	0.375	+43.8pp
`customer_workflow`	0.584	**1.000**	0.688	+41.6pp
`rental_history`	0.361	**0.694**	0.417	+33.4pp
`store_availability`	0.667	**1.000**	0.500	+33.3pp
`good_standing_recommend`	0.583	**0.896**	0.417	+31.3pp
`g_available`	0.722	**1.000**	0.833	+27.8pp
`service_case`	0.694	**0.950**	0.350	+25.6pp
`not_rented`	0.500	**0.750**	0.417	+25.0pp
`return_verify`	0.792	**1.000**	0.417	+20.8pp
`find_customer`	0.834	**1.000**	0.917	+16.6pp
`overdue_report`	0.861	**1.000**	0.334	+13.9pp
`recommend_rating`	**1.000**	**1.000**	0.875	+0.0pp
`not_found`	**1.000**	**1.000**	0.833	+0.0pp
`rental_empty`	**0.667**	**0.667**	0.667	+0.0pp

Tasks combining recommendation with negative filtering or multi-tool workflows (`recommend_category`, `avoid_on_loan`, `upsell_seen`, `customer_workflow`) show gaps between +41.6pp and +50.0pp. Tasks answerable with one simple query (`not_found`, `recommend_rating`) converge at 1.000 across all approaches. The empty-result edge case `rental_empty` scores an identical 0.667 everywhere, suggesting that gracefully reporting "nothing found" is largely model-side behavior independent of interface design.

Under approach B, 15 of 17 tasks are fully solved (mean score 1.0) by at least three of the four models — twelve by all four — indicating that verticalized tools turn task success into a property of the interface rather than of any particular model's SQL skill.

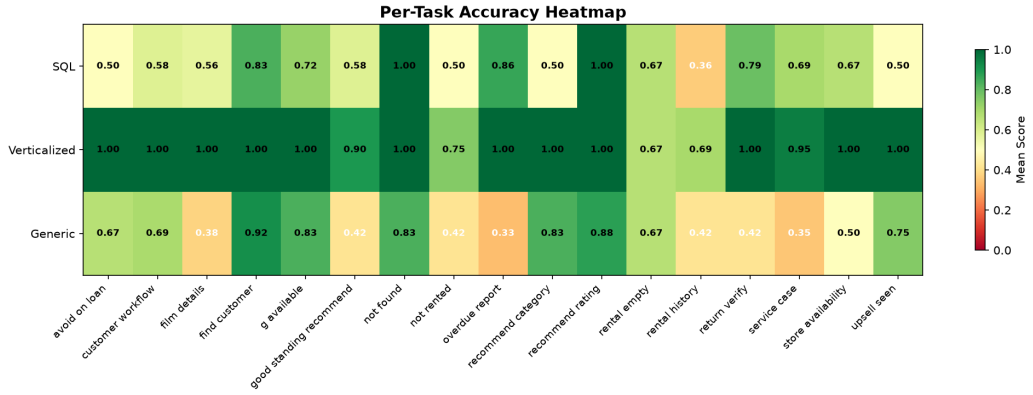


Figure 2: Per-task mean score heatmap. Rows are tasks sorted by Δ(B–A); columns are model $\times$ approach combinations. The verticalized pack achieves near-uniform high scores; raw SQL and the generic pack degrade especially on multi-step workflow and negative-filtering tasks.

Qualitative Observations

Token economy. Approach A embeds the DDL of six tables into every prompt; B and C carry only concise tool descriptions. Measured per-cell means were 3,953 tokens (A), 3,056 (B) and 2,894 (C). The generic pack is marginally cheaper per cell than the verticalized one — but it answers far fewer cells correctly, so the advantage evaporates, and inverts, once responses are weighted by correctness (Section 6.4).

Encapsulation of business rules. Deciding whether an unreturned rental is overdue requires joining `rental` to `film` and comparing `rental_date + rental_duration` against the current time — a rule that models writing raw SQL frequently miss by checking only `return_date IS NULL`. Under approach B the logic lives in `customer_account_summary.sql`:

```
CASE
    WHEN (
        SELECT count(*)::int
        FROM rental r
        JOIN inventory i ON i.inventory_id = r.inventory_id
        JOIN film f ON f.film_id = i.film_id
        WHERE r.customer_id = c.customer_id
          AND r.return_date IS NULL
          AND r.rental_date + f.rental_duration * INTERVAL '1 day' < CURRENT_TIMESTAMP
    ) > 0 THEN 'HAS OVERDUE'
    ELSE 'GOOD STANDING'
END AS standing
```

The domain rule is guaranteed by whoever reviews the SQL file; the model reports it instead of reconstructing it.

Failure modes differ by interface. Raw-SQL failures concentrate in synthesis — malformed joins, missing filters — and grow as models shrink (llama3.2:3b solved 6/51 cells perfectly under A). Generic-pack failures concentrate in routing and workflow composition: C produced 19 zero-score cells across models, versus 7 for A and 3 for B. Under B the argument surface reduces to a few

typed parameters validated server-side, so residual variability stems from routing mistakes rather than query synthesis.

Model tier economics. The decomposition demanded by raw SQL — schema navigation, query synthesis, error recovery — routinely pulls toward larger reasoning models. Domain tools reduce the workload to intent classification and slot filling, so equivalent service becomes reachable by small local models: the study's cheapest correct answer comes from a 3B model with domain tools, not from an 8B model writing SQL.

Threats to Validity

- **Task-aligned scoring.** The verticalized pack and the check-based scorer were developed against the same seventeen tasks. Part of the accuracy gap therefore reflects task-aware tool design — which is intrinsic to verticalization as an engineering practice — but no held-out task set was evaluated, so results should be read as task-aligned rather than as evidence of open-ended generalization.
- **Single small schema.** Sakila exposes six task-relevant tables whose DDL fits entirely in context. This favors approach A if anything; enterprise schemas whose DDL cannot fit in context should widen the gap, but remain untested here.
- **Local small models, single decoding setting.** Four local models between 3B and 8B at temperature 0 with a fixed seed. Frontier cloud models may narrow part of the raw-SQL gap, and robustness across sampling settings is unmeasured.
- **Hardware-relative latency.** Latency compares approaches on fixed local hardware; absolute values do not transfer to other deployments.
- **Missing and excluded runs.** Three cells (0.5%) were lost to a harness fault and two models lacking tool-calling support were excluded before measurement; both facts are documented alongside the frozen results.

Discussion

For practitioners. The benchmark suggests a concrete design sequence for MCP database servers: inventory the recurring questions users actually ask; expose them as named domain operations with rich descriptions and human-readable parameters; push every join and business rule into reviewed SQL; keep an escape hatch for uncovered requests rather than defaulting to raw SQL access. Treat tool definitions as API contracts — versioned, reviewed, and covered by tests; the same rule sets used for scoring double as regression tests for packs.

Relation to text-to-SQL. The pattern does not compete with text-to-SQL research but repositions it. NL-to-SQL remains the right interface for exploration, ad-hoc analytics and prototyping; production serving paths benefit from bounded operations whose semantics are guaranteed server-side. A pragmatic deployment can offer both surfaces to different audiences under different credentials.

When generic interfaces suffice. Generic SQL tools remain reasonable where data is exploratory, users are expert analysts, and consequences are low. The pattern targets the complementary regime: recurring operational questions, non-expert users, and autonomous agents acting without human review of each query.

Ecosystem implications. Because packs are declarative artifacts, they invite infrastructure that does not exist for prompt-side fixes: code review workflows, portability across database engines, role-based visibility filtering, and shared registries of tested domain packs. Security posture also changes qualitatively: with no arbitrary-query surface, indirect injection [17] can manipulate routing but cannot rewrite the queries themselves.

Related Work

Agentic tool use. ReAct [4] established interleaved reasoning and acting as the dominant agent loop; Toolformer [5] showed models can self-supervise tool calls, and Gorilla [6] and ToolLLM [7] scaled tool selection over large API corpora. Function-calling leaderboards such as BFCL [8] evaluate tool-selection accuracy directly. This line treats the tool surface as given; our results indicate that surface design itself is a first-order variable worth benchmarking.

Text-to-SQL. Spider [9] initiated large-scale cross-domain evaluation, followed by harder benchmarks such as BIRD [10]; DIN-SQL [11] and DAIL-SQL [12] pushed accuracy through decomposition and prompt engineering. These systems optimize generation quality, whereas the Domain-Oriented Tooling Pattern removes generation from the serving path entirely; the two approaches are complementary (Section 7).

Abstraction layers for data access. Object-relational mappers abstract SQL behind language objects for imperative code; REST [3] replaced unconstrained RPC with bounded resources; semantic layers in analytics stacks define governed metrics. Domain-driven design [2] supplies the organizational principle. The pattern applies this lineage to probabilistic clients, adding requirements ORMs never faced: descriptions must steer model routing, and parameter contracts must tolerate natural-language inputs.

Small open models. Llama 3 [13], Qwen2.5 [14], Phi-3 [15] and Gemma 2 [16] made capable 3B–8B models runnable on workstations. Prior work typically measures these models on generation-heavy tasks; our benchmark measures how interface design shifts which tier suffices for reliable tool use — the empirical core of Model Demotion.

MCP ecosystem and agent reliability. Recent work addresses multi-agent interoperability [18], the sustainability economics of agentic systems [19], and prompt-injection defenses combining nested learning with semantic caching [17]. Our contribution is orthogonal: it hardens the boundary between agent and database itself.

Conclusion and Future Work

We proposed the Domain-Oriented Tooling Pattern: expose databases to LLM agents as curated domain operations rather than generic SQL execution. The pattern rests on three invariants — encapsulated data access, deterministic business rules, declarative tool definition — and yields Model Demotion: when synthesis gives way to tool selection, smaller models serve requests reliably. MCP Blueprint demonstrates that the pattern can be implemented declaratively, keeping protocol infrastructure separate from domain knowledge.

A public reproducibility benchmark compared three server designs across four local models and seventeen tasks. The verticalized pack reached 0.939 pooled mean score versus 0.666 for raw SQL and 0.605 for a generic thin-tool pack; every model gained, the smallest most of all ($0.583 \rightarrow 0.929$);

cost per correct answer fell by factors of roughly 2–12×; and a superficially similar generic pack scored below raw SQL, isolating tool design — not tool existence or model scale — as the decisive factor. All artifacts are public and the run is fully reproducible.

Future work proceeds along five lines:

1. **Cross-domain replication** with additional schemas, organically collected user requests, and held-out task protocols separating pack authoring from evaluation.
2. **Frontier and hosted models** to test whether the raw-SQL gap narrows at scale, and temperature sweeps for robustness characterization.
3. **Automated pack authoring**: generating candidate YAML/SQL definitions from validated views and OpenAPI specifications, with human review as the acceptance gate.
4. **Governance features**: role-aware tool visibility, audit trails, and pack registries with compatibility certification.
5. **Beyond relational stores**: extending the declarative pack model to document stores and vector retrieval systems.

Declarative Tool Specification

A complete domain tool consists of one YAML file and one SQL file. The YAML declares the tool name, its natural-language description (the primary routing signal for the model), typed parameters, the backing SQL file, and cache behavior:

```yaml
# packs/sakila/tools/film_stock.yaml
name: film_stock
description: >-
  Per-store stock for a film found by title.  One call returns one row per
  store with the total copies, the copies currently available (not on loan),
  plus the film's rating and length in minutes.  The title is matched
  case-insensitively as a substring.  Pass store_id only to filter to a
  single store.  Use this when a customer asks how many copies of a film are
  available or whether it is in stock at a store.
parameters:
  title:
    type: string
    required: true
    description: Film title or a substring of it, matched case-insensitively.
  store_id:
    type: integer
    required: false
    default: null
    description: Optional store identifier to filter the result to one store.
sql: ../sql/film_stock.sql
cache:
  ttl: 30
```

SQL files remain isolated from application code. Values always arrive as bound placeholders (`%(title)s`, `%(store_id)s`), preventing injection, and the business rule — a copy is available

when no open rental references it — is computed inside the query itself rather than by the model. Optional parameters are handled with a lightweight template conditional (`{% if store_id %}`): when the model omits `store_id`, the filter simply disappears from the rendered SQL, so one definition serves both filtered and unfiltered calls.

```sql
-- packs/sakila/sql/film_stock.sql
SELECT f.film_id,
       f.title,
       f.rating::text  AS rating,
       f.length,
       s.store_id,
       COUNT(i.inventory_id)  AS total_copies,
       COUNT(i.inventory_id) FILTER (
           WHERE NOT EXISTS (
               SELECT 1
               FROM rental r
               WHERE r.inventory_id = i.inventory_id
                 AND r.return_date IS NULL
           )
       )  AS available
FROM film f
JOIN inventory i ON i.film_id = f.film_id
JOIN store s ON s.store_id = i.store_id
WHERE f.title ILIKE '%%' || %(title)s || '%%'
{% if store_id %}
  AND s.store_id = %(store_id)s
{% endif %}
GROUP BY f.film_id, f.title, f.rating, f.length, s.store_id
ORDER BY f.film_id, s.store_id
LIMIT 50;
```

Because both files are declarative artifacts, the tool can be reviewed, tested, versioned and ported to another database engine without touching protocol code.

Benchmark Tasks

#	Task id	Prompt
1	`find_customer`	Find the customer whose last name is Smith. Report the customer's full name and customer ID.
2	`rental_history`	Mary Smith wants to know what she has rented. What films has she rented, and does she currently have any rentals outstanding?

#	Task id	Prompt
3	`good_standing_recommend`	Check whether customer Maria Miller has any overdue rentals. If she is in good standing, recommend two Sci-Fi movies.
4	`overdue_report`	Check whether customer Tammy Sanders has any overdue rentals. If she does, list the films she still has to return.
5	`recommend_category`	Recommend three popular Family movies for a family movie night.
6	`recommend_rating`	Recommend two movies rated G, suitable for all ages.
7	`film_details`	Tell me about the movie 'Goodfellas Salute': its rating, its length in minutes, and how many copies are available.
8	`avoid_on_loan`	Customer Tammy Sanders is at the counter right now. Recommend one Science Fiction movie that she is not currently renting.
9	`not_found`	Find the customer whose last name is Doe.
10	`customer_workflow`	A customer named Jennifer Davis is asking about her account. Check her rental situation and then recommend a Documentary movie she might enjoy.
11	`upsell_seen`	Customer Kelly Torres enjoyed the Science Fiction movies she has rented before. Recommend two other popular Science Fiction movies that she has NOT rented before.
12	`return_verify`	Customer Mary Smith says she has returned everything she rented. Verify from the records: does she still have any rentals on loan? If so, list the film titles.
13	`store_availability`	A customer at Store 2 is asking for 'Goodfellas Salute'. How many copies are available, and is it in stock at Store 2?
14	`g_available`	Recommend a G-rated movie that is currently available in stock for a family movie night.

#	Task id	Prompt
15	`service_case`	Customer Tammy Sanders is calling about late fees. Check her account: which films are currently on loan, which of those are overdue, and what is her home store so the store can reach out to her?
16	`rental_empty`	Show me the rental history for customer ID 9999.
17	`not_rented`	Which Science Fiction movies has Mary Smith NOT rented before?

References

1. Anthropic. (2024). *Model Context Protocol Specification*. https://modelcontextprotocol.io
2. Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.
3. Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures* (Doctoral dissertation, University of California, Irvine).
4. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). *ReAct: Synergizing Reasoning and Acting in Language Models*. ICLR. arXiv:2210.03629.
5. Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). *Toolformer: Language Models Can Teach Themselves to Use Tools*. NeurIPS. arXiv:2302.04761.
6. Patil, S. G., Zhang, T., Wang, X., & Gonzalez, J. E. (2023). *Gorilla: Large Language Model Connected with Massive APIs*. arXiv:2305.15334.
7. Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., Lin, Y., Cong, X., Tang, X., Qian, B., et al. (2023). *ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs*. arXiv:2307.16789.
8. Yan, F., Mao, H., Liu, C. C., Tang, K., Lou, R., Li, H., Yin, W., Yu, P. S., & Zhang, T. (2024). *Berkeley Function Calling Leaderboard*. arXiv:2408.04682.
9. Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., et al. (2018). *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task*. EMNLP. arXiv:1809.08887.
10. Li, J., Yuan, Y., Zhang, G., Yu, B., Li, L., Li, T., et al. (2023). *Can LLM Already Serve as a Database Interface? A BI Benchmark on Complex SQLs (BIRD)*. NeurIPS Datasets and Benchmarks. arXiv:2305.03111.
11. Pourreza, M., & Rafiei, D. (2023). *DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction*. NeurIPS. arXiv:2304.11015.
12. Gao, D., Wang, H., Li, Y., Sun, X., Qian, Y., Bian, B., et al. (2024). *Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation (DAIL-SQL)*. PVLDB 17(5). arXiv:2308.15363.
13. Meta AI. (2024). *The Llama 3 Herd of Models*. arXiv:2407.21783.
14. Qwen Team. (2024). *Qwen2.5 Technical Report*. arXiv:2412.15115.

15. Abdin, M., et al. (2024). *Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone*. arXiv:2404.14219.
16. Gemma Team. (2024). *Gemma 2: Improving Open Language Models at a Practical Size*. arXiv:2408.00118.
17. Gosmar, D., & Dahl, D. A. (2026). *Prompt Injection Mitigation with Agentic AI, Nested Learning, and AI Sustainability via Semantic Caching*. IFIP AIAI, Springer. doi:10.1007/978-3-032-30805-4_21
18. Gosmar, D., Dahl, D. A., Coin, E., & Attwater, D. (2024). *AI Multi-Agent Interoperability Extension for Managing Multiparty Conversations*. arXiv:2411.05828.
19. Gosmar, D., Pallotta, A. C., & Zenezini, G. (2025). *Agentic AI Sustainability Assessment for Supply Chain Document Insights*. arXiv:2511.07097.
20. Bogliolo, B. (2026). *MCP Blueprint: Declarative Framework for Model Context Protocol Servers*. https://github.com/meob/mcp-blueprint
21. Bogliolo, B. (2026). *MCP Blueprint Benchmark: Reproducible Harness for MCP Server Design Evaluation*. https://github.com/meob/mcp-blueprint-benchmark