MADBench - Measuring Agentic DBMS

Quantifying the Protocol Tax of MCP-Mediated Query Workloads

by

Yaseen Ahmed

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2026

© Yaseen Ahmed 2026

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Large language model agents increasingly access relational databases through the Model Context Protocol (MCP), a tool-calling standard in which the model discovers tool definitions from a middleware server, issues calls through it, and consumes query results serialized as text into its context window. Under this architecture, every database interaction pays costs that classical database clients do not: tool discovery, server lifecycle, result serialization, transport, and context-window consumption. Existing agent benchmarks report end-to-end task scores and therefore cannot attribute these costs to the layer that causes them.

This thesis presents MADBench, a benchmark that measures and decomposes this overhead, termed the Protocol Tax: end-to-end interaction time minus raw database execution time. OpenTelemetry instrumentation splits the tax into seven protocol phases (discovery, lifecycle, handover, serialization, parsing, transport, and teardown), with the unattributed residual held below 5% of wall-clock time on 98.7% of runs. Two measurement tracks separate protocol effects from model effects. Track A replays fixed TPC-H queries through three open-source MCP servers over PostgreSQL, MySQL, and DuckDB at scale factors 1 through 10, under paired warm-start and cold-start conditions, collecting 5,378 traces with no model in the loop. Track B runs a fixed LLM agent over 30 pre-registered natural-language questions on the same server-backend grid, yielding 1,080 attempts scored against gold SQL results.

Three findings emerge. First, Protocol Tax spans three orders of magnitude on identical queries, from 10^{-3} of database execution time for an in-process engine to $3\times$ it for a subprocess server on PostgreSQL; the dominant cost is the server's process model, not its wire format. Second, protocol speed does not predict agent success: the server with the lowest Protocol Tax ranks last in agent completion, and DuckDB carries $112\times$ less protocol overhead than the client-server backends yet finishes mid-pack. Third, completion is instead governed by result-payload compactness and schema-metadata quality; one server-backend pairing collapses from 61.1% to 17.8% completion between scale factors when verbose query results overflow the model's context window. The benchmark, pre-registered workload, and

full trace corpus are released to support reproduction and extension.

Acknowledgements

I would like to thank my supervisor, Professor M. Tamer Özsu, for his guidance and feedback on this thesis. I also thank my family for their constant support.

AI Disclosure

The author used Claude (Anthropic) to assist with outlining, diagram production, and editorial review. All results, analysis, and final content are the author's own.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements v

AI Disclosure vi

List of Figures xiii

List of Tables xiv

List of Abbreviations xvi

1 Introduction 1

1.1 The Middleware Problem 1

1.2 Protocol Tax in Agentic DBMS 3

1.3 MCP and DBMS Integration 5

1.4 Research Questions 6

1.5 Thesis Contributions 7

1.5.1 Protocol Tax as a Decomposable Systems Construct 7

1.5.2 Two-Tier Attribution Framework 8

1.5.3 Two-Track Experimental Design . . . 8
1.5.4 MADBench . . . 9
1.6 A Motivating Example . . . 9
1.7 Approach and Methodology Overview . . . 12
1.7.1 Workload Design . . . 12
1.7.2 Systems Under Test . . . 12
1.7.3 Instrumentation . . . 12
1.7.4 Analysis Framework . . . 13
1.8 Scope and Limitations . . . 13
1.8.1 In Scope . . . 13
1.8.2 Out of Scope . . . 13
1.8.3 Attribution Boundaries . . . 14
1.9 Thesis Organization . . . 14

2 Literature Review **16**
2.1 Middleware Friction in Agentic Data Systems . . . 16
2.2 From Data Shipping to Knowledge Shipping . . . 17
2.2.1 Classical Data Movement and Interface Stability . . . 17
2.2.2 LLM Agents as Query Producers . . . 18
2.2.3 Knowledge Shipping as a Third Mode . . . 18
2.2.4 Schema Linking as a Dominant Failure Source . . . 19
2.2.5 The Missing Metric Framework . . . 19
2.3 MCP Architecture and Ecosystem Heterogeneity . . . 20
2.3.1 Tool Granularity and Discovery Overhead . . . 21
2.3.2 Schema Depth, Metadata Richness, and Grounding Fidelity . . . 22
2.3.3 Connection Lifecycle and Serialization Boundaries . . . 23
2.3.4 Safety Models and the Security-Performance Boundary . . . 24
2.4 MCP and Agent Benchmarks . . . 27

2.4.1 Task Completion Benchmarks and Their Diagnostic Limits 27
2.4.2 Security Benchmarks and Exploit Attribution 28
2.4.3 The Attribution Blind Spot in DBMS Couplings 28
2.4.4 Toward DBMS-Style Control for MCP Measurement 29
2.5 Schema Grounding and Metadata Quality 30
2.5.1 Schema Linking as the Primary Bottleneck 30
2.5.2 Error Taxonomies and Generation-Side Methods 31
2.5.3 Metadata Quality as a Systems Variable 31
2.5.4 Complexity-Conditioned Effects and Category-Aware Reporting . . 32
2.6 Workload Design for Middleware Benchmarking 33
2.6.1 Workload Control and Attribution Strength 33
2.6.2 Payload Sensitivity and the TPC-H and SQLyzr Pairing 33
2.6.3 Cross-Backend Comparability and Reporting Conventions 34
2.7 Instrumentation and Measurement Methodology 35
2.7.1 Distributed Tracing as Methodological Baseline 35
2.7.2 Two-Track Design and Metric Attribution 36
2.7.3 Operational Requirements for Reliable Attribution 38
2.8 Summary and Research Gap 39

3 MADBench: Workload, Dataset, and Metrics 44
3.1 Dataset 44
3.1.1 TPC-H Schema 44
3.1.2 Data Generation and Scale Factors 45
3.2 Workload 45
3.2.1 Track A: Direct SQL Workload 46
3.2.2 Track B: agentic question workload 47
3.3 Evaluation Metrics 49
3.3.1 Protocol Tax Decomposition 49
3.3.2 Implementation and Coupling Metrics 55

4 Benchmarking Framework **58**
4.1 Framework overview . . . 58
4.2 Systems under test . . . 59
4.2.1 MCP server selection . . . 59
4.2.2 DBMS backends and measured grid . . . 60
4.2.3 Hardware and software environment . . . 60
4.3 Instrumentation schema . . . 61
4.3.1 Span hierarchy and timing boundaries . . . 61
4.3.2 Database-time reference . . . 62
4.3.3 Span attributes . . . 62
4.3.4 Exclusion and calibration rules . . . 63
4.3.5 Calibration procedure . . . 63
4.4 Experimental protocols . . . 64
4.4.1 Track A: controlled measurement . . . 65
4.4.2 Paired cold and warm runs . . . 66
4.4.3 Track B: agent-present measurement . . . 66
4.5 Correctness and failure classification . . . 67
4.5.1 Comparison semantics . . . 68
4.5.2 Shape drift . . . 68
4.5.3 Failure-mode analyzer . . . 68
4.6 Analysis procedures . . . 68
4.6.1 Track A analysis . . . 69
4.6.2 Track B analysis . . . 69
4.6.3 Cross-track analysis . . . 69
4.6.4 Statistical controls . . . 70
4.7 Cost control and reproducibility . . . 71
4.7.1 Cost tracking . . . 71
4.7.2 Reproducibility artifacts . . . 71
4.8 Summary . . . 72

5 Experiments and Results **73**
5.1 Experimental corpus . . . 73
5.1.1 Executed runs . . . 74
5.1.2 Protocol-tax accounting . . . 75
5.2 Track A: controlled protocol measurements . . . 76
5.2.1 Overall Protocol Tax Ratio . . . 76
5.2.2 Phase attribution . . . 77
5.2.3 Effect of scale factor . . . 77
5.2.4 Trace validity . . . 78
5.2.5 Index sensitivity . . . 79
5.3 Track B: agent-present results . . . 79
5.3.1 Completion at SF1 . . . 80
5.3.2 Failure modes . . . 80
5.3.3 Scale-factor robustness . . . 81
5.3.4 Tool surface and tool-call volume . . . 83
5.3.5 Cost and token use . . . 84
5.4 Workload complexity and verifier results . . . 85
5.4.1 C2 and C5 completion . . . 85
5.4.2 Interaction cost by category . . . 87
5.4.3 Verifier diagnostics . . . 87
5.4.4 Per-query difficulty . . . 88
5.5 Cross-track analysis . . . 89
5.5.1 Rank-order comparison . . . 89
5.5.2 Retry proxy . . . 90
5.5.3 Statistical detectability . . . 90
5.6 Scope and limitations . . . 90
5.6.1 Metric coverage . . . 92
5.7 Summary of findings . . . 93
5.7.1 Answers to the research questions . . . 93

6 Conclusion and Future Work **95**

6.1 Conclusion . . . 95

6.1.1 Answers to the research questions . . . 95

6.1.2 Contributions . . . 98

6.2 Future work . . . 98

6.2.1 Broader agent evaluation . . . 98

6.2.2 Large-result failures . . . 99

6.2.3 Measurement coverage . . . 99

6.2.4 Other protocol designs . . . 99

6.3 Final remarks . . . 100

References **101**

Appendix A Workload Queries **110**

A.1 Track A: TPC-H reference queries . . . 110

A.2 Track B: SQLyzr-C2 workload (15 queries) . . . 113

A.3 Track B: SQLyzr-C5 workload (15 queries) . . . 114

A.4 Track B: gold SQL for all thirty questions . . . 115

A.5 Track B: agent system prompt . . . 119

Glossary **121**

List of Figures

3.1 End-to-end timing model for a single MCP-mediated database interaction. The nine timing terms are the eight measured phases (T_{disc}, T_{life}, T_{hand}, T_{DB}, T_{ser}, T_{parse}, T_{xport}, T_{teardown}) plus the residual ϵ. Protocol Tax ΔP is the sum of all measured phases except T_{DB}. Transport (T_{xport}) is shown as a dashed bar because it may temporally overlap with handover and parse phases. . . 51

5.1 Verifier-confirmed completion for all six cells at SF1 and SF3. The main change is the `dbhub`×PostgreSQL cell at SF3. 82

5.2 C2 and C5 completion at SF1. All six cells show the same endpoint ordering. 86

5.3 Verifier diagnostic classes across all 1,080 Track B attempts. 88

List of Tables

1.1 Illustrative cost breakdown for a single MCP-mediated analytical query. The DBMS engine still accounts for a small fraction of the total interaction time relative to protocol tax. 11

2.1 DBMS-facing MCP server design axes (public documentation snapshot, March 2026). 22

3.1 TPC-H schema at SF1 (Transaction Processing Performance Council, 2022). Row counts grow with scale factor except for `region` and `nation`. 45

3.2 TPC-H queries used in Track A and their result cardinalities in the executed workload. 46

3.3 SQLyzr categories used to stratify the Track B workload (Abedini, 2026). . 47

3.4 Timing phases used in the Protocol Tax decomposition. 50

3.5 Implementation Metrics and their main developer levers. 56

3.6 Coupling Metrics and their main sources of variation. 57

4.1 MCP servers used in the study. Tool counts are the counts exposed to the client during execution. 59

4.2 Server-backend combinations used by MADBench. 60

4.3 Main attributes recorded by the trace pipeline. 63

4.4 Executed experimental configuration. 64

5.1 Server–backend combinations used in both tracks. 74

5.2 Executed experimental corpus. 75

5.3 Track A Protocol Tax Ratio by server–backend combination. The middle column is the median of cell medians. The range covers query, scale factor, and cold or warm state. 76
5.4 Median phase duration for PostgreSQL Q6 at SF1, warm state (ms). Tail covers teardown or the unattributed interval after the last child span. . . . 77
5.5 Median Track A Π by scale factor. Each value summarizes query and cold or warm cell medians within one coupling. 78
5.6 Effect of the `l_shipdate` index on PostgreSQL Q6. Π includes the complete protocol path. 79
5.7 Track B results at SF1. Each cell contains 90 attempts. 80
5.8 Track B outcome counts at SF1. CLE denotes call-limit exceeded. 81
5.9 Verifier-confirmed completion at SF1 and SF3. Each rate is based on 90 attempts. 82
5.10 Tool-call counts per Track B attempt at SF1 and SF3. 83
5.11 Cost per correct answer at SF3. 84
5.12 C2 and C5 completion at SF1. Each category contains 45 attempts per cell. 85
5.13 Mean cumulative input-token ratio, C5 divided by C2, at SF1. This is a proxy for context growth, not the calibrated IM-8 metric. 87
5.14 Verifier diagnostics across both Track B scale factors. 87
5.15 Queries at the lower end of the pooled completion distribution. Each query has 36 attempts across six cells, two scale factors, and three repetitions. . . 89
5.16 Server-level join of Track A median Π and Track B mean completion. . . . 89
5.17 Main limits of the reported experiments. 91
5.18 Coverage of the Implementation and Coupling metrics. 92
5.19 Main experimental findings. 93

A.1 Track B category C2 workload (15 hand-written natural-language queries pre-registered before any LLM run). `target_subcat` names the SQLyzr subcategory the query was designed to elicit. 113
A.2 Track B category C5 workload (15 hand-written natural-language queries pre-registered before any LLM run). `target_subcat` names the SQLyzr subcategory the query was designed to elicit. 114

List of Abbreviations

MCP Model Context Protocol

LLM Large Language Model

DBMS Database Management System

RPC Remote Procedure Call

JSON JavaScript Object Notation

SQL Structured Query Language

DDL Data Definition Language

NL-to-SQL Natural Language to SQL

JDBC Java Database Connectivity

ODBC Open Database Connectivity

TNS Transparent Network Substrate

TDS Tabular Data Stream

gRPC Google Remote Procedure Call

TPC-H Transaction Processing Performance Council Benchmark H

TPC-C Transaction Processing Performance Council Benchmark C

OLAP Online Analytical Processing

OLTP Online Transaction Processing

SF Scale Factor

PER Payload Expansion Ratio

RSS Resident Set Size

OTel OpenTelemetry

RL Reinforcement Learning

RAG Retrieval-Augmented Generation

VLDB Very Large Data Bases

SIGMOD Special Interest Group on Management of Data

PVLDB Proceedings of the VLDB Endowment

Chapter 1

Introduction

1.1 The Middleware Problem

A modern analytical database engine can finish a TPC-H query in milliseconds. At scale factor 1 (∼1 GB) on a single thread, systems modeled after MonetDB/X100 (Boncz et al., 2005) and HyPer (Kemper and Neumann, 2011) execute representative queries in tens to hundreds of milliseconds. Even the most join-heavy queries complete in under 250 ms (Kersten et al., 2018), and DuckDB carries the same vectorized-execution lineage into embedded analytical workloads (Raasveldt and Mühleisen, 2019). These performance gains reflect decades of investment in columnar storage (Stonebraker et al., 2005), vectorized execution and cache-conscious algorithms (Boncz et al., 2005), and the standardized workloads that drive query-engine optimization (Transaction Processing Performance Council, 2022). For analytical workloads at moderate scale, the database engines are fast.

The client architecture surrounding these engines is changing. Rather than issuing SQL through hand-written application logic, many modern systems place a large language model (LLM) between the user and the database management system (DBMS). Some interfaces treat the model as a natural-language-to-SQL translator that converts a request into an executable statement. Recent agentic systems embed the LLM within an autonomous agent that reasons across multiple interactions, maintains state, invokes external tools, and refines its actions (Wang et al., 2024). Xi et al. (2025) formalize this architecture as a three-component design: a reasoning module (the "brain") that owns memory and drives planning, a perception module that connects the agent to its environment, and an action module that executes decisions through tool calls. Unlike a translator, which maps one input to one output, an agent iterates, backtracks, and adapts its plan as new

information arrives. Agents built this pattern now operate in software engineering and natural science (Wang et al., 2024) as well as in enterprise tasks across data analysis, finance, and human resources (Xu et al., 2025). They are evaluated on workloads that require querying databases, navigating web interfaces, and managing multi-step workflows (Liu et al., 2024; Xu et al., 2025).

DBMS access is a prominent application domain for these agents. Organizations use LLMs to translate natural-language questions into SQL and explore unfamiliar schemas (Shi et al., 2026; Liu et al., 2026). Liu et al. (2026) frame this shift as *agentic speculation*. More specifically, they describe it as a workload pattern in which agents issue rapid, exploratory queries across multiple rounds, probing data sources before converging on an action plan. Fernandez et al. (2023) argue that LLMs will disrupt data management on two fronts, surpassing the automation ceiling on the "hard database problems" (entity resolution, schema matching, data discovery, query synthesis) and eroding the boundary between predictive models and information retrieval systems. This approach allows a user who cannot write SQL can still ask complex analytical questions. It also introduces a new computational layer between the user's intent and the engine's response.

LLM-mediated DBMS access carries a measurable computational cost. Akillioglu et al. (2025) report that processing roughly 17,000 rows through an in-DBMS LLM inference pipeline took about five hours of local inference running on an NVIDIA A100 GPU, placing the vast majority of wall-clock time in the model layer rather than the relational operations. Bian et al. (2025) decompose latency across agentic pipeline stages and find that more than half of wall-clock time can originate outside model inference. Web environment overhead alone reaches more than half of end-to-end duration, while API response times under fixed input prompts and controlled output lengths vary significantly across dates and geographic regions. These studies show that environment and middleware costs can match or exceed model inference time in real settings, making the interaction layer worth optimizing.

For LLM agents, the dominant cost typically lies along the execution path between the model and the external system rather than in the model itself. Instead of submitting SQL through a native driver and receiving a binary result set, an agent calls a tool to discover available tables and reads schema metadata returned as text. It then reasons about column names, data types, and join paths before synthesizing a query, sending it through a protocol layer, and receiving results serialized as JSON. A failed query triggers another attempt. Each of these steps consumes tokens, time, and network round-trips that are not present in the traditional client-driver path. This overhead is a largely unmeasured middleware engineering problem, much different than model intelligence and database performance.

The scale of deployment makes this measurement gap urgent. The Model Context

Protocol (MCP), an open protocol originally developed by Anthropic for connecting LLM applications to external tools and data sources, has grown to thousands of server implementations in under a year (Anthropic, 2025; Hou et al., 2026). DBMS-specific servers exist for PostgreSQL, MySQL, SQL Server, DuckDB, Oracle, and dozens of other systems, built by independent developers with widely varying design philosophies. Agent capability on these tasks remains limited. AgentBench evaluates LLMs across numerous interactive environments and reports that commercial models average significantly higher scores than open-source alternatives, with task-limit exhaustion the main failure mode (Liu et al., 2024); TheAgentCompany finds that even the strongest model completes only 30% of simulated workplace tasks autonomously (Xu et al., 2025). These limitations increase middleware costs because agents retry, backtrack, and issue redundant tool calls, each of which passes through the protocol layer. The resulting overhead affects deployment architecture, cost, and reliability.

1.2 Protocol Tax in Agentic DBMS

This thesis introduces the concept of **Protocol Tax**, defined as all latency and resource consumption in an agentic database interaction except raw database execution time. Rather than treating protocol tax as a single number, the thesis decomposes it into seven distinct phases:

1. **Discovery.** The agent queries the MCP server for available tools and schema metadata. This consumes tokens and time before any data-bearing call executes.
2. **Lifecycle.** This phase covers connection setup, authentication, and session management. Some servers require multi-step handshake sequences, while other maintain persistent tools.
3. **Handover.** The transition from agent reasoning to tool invocation and back. Each handover crosses a process or network boundary.
4. **Serialization.** Conversion of DBMS results from native binary wire format to JSON text for protocol transport. This cost grows linearly with result set size.
5. **Parsing.** Client-side deserialization of JSON responses and materialization into structures the LLM can consume. For large payloads, parse time can be substantial (Langdale and Lemire, 2019), and text-based encodings carry significant size overhead relative to binary alternatives (Viotti and Kinderkhedia, 2022).

6. **Transport.** Network transfer of protocol messages between host, client, and server components.

7. **Teardown.** This phase includes the wait for subporcess temrination and the flushing of buffered pipes after the last data-bearing call. For servers with an irregular cleanup process, this interval is more significant. Chapter 3 reports a median 28% of T_{total} on the Node.js reference server.

These phases are introduces because the agent communicates through a standardized protocol layer rather than through a traditional database driver. Most are absent or negligible in traditional DBMS access, where mature wire protocols use binary encodings, persistent connections, and streaming result transfer to minimize the per-query overhead (PostgreSQL Global Development Group, 2025; Oracle Corporation, 2025a; Microsoft Corporation, 2026). MCP replaces that machinery with text-oriented, schema-heavy JSON-RPC message exchange (Anthropic, 2025; JSON-RPC Working Group, 2010). By comparison, MCP uses text-oriented and schema-heavy messages that rely on JSON-RPC (Anthropic, 2025; JSON-RPC Working Group, 2010).

Protocol tax is most pronounced in DBMS workloads. Many MCP tools return short responses, such as weather lookups or search results. On the other hand DBMS queries can return thousands to millions of rows. Schema metadata can also span hundreds of tables with rather complex foreign-key relationships. Each round-trip also involves binary-to-text conversion that scales with output size. Agents may require several rounds. Ding et al. (2025) measure MCP interactions across nine LLMs and find that output tokens are a small fraction of each exchange.

Existing systems provide little visibility into the sources of significant protocol overhead in DBMS interactions. For example, an MCP-mediated query that takes 30 seconds offers no breakdown across model inference, schema discovery, serialization, connection setup, or client-side JSON parsing, and current benchmarks cannot supply one. Some decompose success into evaluator categories such as format compliance and static versus dynamic correctness (Luo et al., 2025a). Others measure task pass rates and token efficiency by server scope and call dependency (Guo et al., 2026). Neither approach attributes latency to individual pipeline stages and isolates the middleware components that cause overhead in DBMS interactions. AI-NativeBench shows that such attribution is feasible for general agentic workflows and reports inference-dominant latency profiles (Wang et al., 2026), but it does not target the DBMS-specific coupling spans where serialization, schema transfer, and lifecycle overhead concentrate.

Heterogeneity introduces further complexity into the problem. As a framework, MCP standardizes message format but not implementation policy. Two fully compliant DBMS

servers can produce very different latency and token footprints for the same query. One may expose two tools while another exposes dozens. Some returns CSV, whereas others return JSON objects with repeated key names on every row. Schema exposure also varies. One server may hide its schema behind templates, while another includes full metadata in every discovery response. These are developer choices rather than protocol constraints (Anthropic, 2025; Hou et al., 2026). Current benchmarks compare servers as monolithic entities without separating developer-controlled effects from those attributed to the DBMS itself.

Luo et al. (2025b) evaluate MCP servers on web-search and DBMS tasks and conclude that MCP-mediated tool use "do[es] not consistently demonstrate marked improvements over non-MCP approaches, such as function calls," with accuracy varying from under 15% to 64% across servers on equivalent tasks. Bian et al. (2025) report that environment interaction latency can rival inference time. These findings point to protocol overhead as a possible contributor to end-to-end latency, but no existing benchmark provides the decomposed, DBMS-specific measurement needed to act on it.

1.3 MCP and DBMS Integration

The Model Context Protocol formalizes a client-host-server architecture for tool invocation (Anthropic, 2025). Hosts are LLM applications that initiate connections and orchestrate conversations. Clients are connectors within the host that mediate between host and server, routing tool calls and returning results. Servers expose executable tools alongside the metadata required for invocation. Communication uses JSON-RPC over transports such as `stdio` and Streamable HTTP (Anthropic, 2025; JSON-RPC Working Group, 2010).

In database access, an MCP server translates between the protocol's text-oriented message format and the DBMS's native wire protocol. For example, a PostgreSQL MCP server receives a JSON-RPC tool call, translates its arguments to SQL, and sends the query over PostgreSQL's binary frontend/backend protocol. After receiving the typed binary-encoded results, the server converts them to JSON, wraps them in an MCP response envelope, and returns the response to the client (PostgreSQL Global Development Group, 2025). A DuckDB MCP server follows a different path. The database may run in-process, eliminating the network boundary while still requiring conversion from DuckDB's columnar internal format to JSON text (DuckDB Foundation, 2025). Each conversion boundary is a potential source of overhead, and the number of boundaries varies with the backend architecture.

MCP has been adopted rapidly. Survey work documents thousands of server implementations, broad integration with frameworks, and adoption by companies like Anthropic and OpenAI (Hou et al., 2026). For DBMSs specifically, implementations span PostgreSQL, MySQL, SQL Server, DuckDB, Oracle, and others, built by independent developers with widely varying design philosophies. Ecosystem surveys place the mean server near a dozen tools, with roughly half exposing four or fewer (Bloomberry, 2026); DBMS servers range from single-tool implementations to ones containing dozens of tools (Bentham, 2024). Section 2.3 quantifies this distribution.

A shared protocol allows for systematic performance comparisons because servers implementing the same specification can be tested with the same workloads and instrumentation. As Section 1.2 notes, however, MCP prescribes message exchange but leaves schema exposure, tool-description verbosity, connection policy, and output format to the implementation (Anthropic, 2025). Servers that are equivalent at the protocol level can therefore behave very differently in practice.

MCP is not the only agent interoperability standard. Ehtesham et al. (2025) survey four emerging standards (MCP, ACP, A2A, and ANP) and characterize their architectural differences. MCP uses a centralized JSON-RPC design optimized for tool integration, while A2A and ANP target peer-like agent coordination with different transport and discovery mechanisms (Ehtesham et al., 2025). Because MCP relies on JSON-RPC, it inherits both the size penalty of text-based encodings relative to binary wire formats (Viotti and Kinderkhedia, 2022) and the parsing cost that JSON imposes at scale (Langdale and Lemire, 2019), a tradeoff relevant to data-intensive use cases. For database access specifically, MCP is the protocol that practitioners encounter, and it is the protocol where measurement infrastructure is most needed.

The shared protocol makes measurement possible because different servers can be tested against the same workloads. At the same time, the implementation freedom allowed by MCP makes such measurement necessary, since protocol compliance alone cannot predict performance.

1.4 Research Questions

Protocol overhead in agentic DBMS access is tangible, but it has not been broken down phase by phase or separated from differences among server implementations. This thesis addresses this gap through four research questions

RQ1: Scale transition. Under OLAP-scale workloads, does MCP protocol overhead exceed raw DBMS execution time, and at what scale factor does this transition occur?

RQ2: Attribution. How much of the observed performance variation across MCP DBMS servers is attributable to implementation choices (tool design, schema policy, serialization format) versus coupling effects inherent to the DBMS-protocol integration?

RQ3: Predictive validity. Do implementation-level differences measured under controlled, agent-free conditions predict real-world differences in agent retry behaviour, context-window decay, and task completion under natural-language uncertainty?

RQ4: Methodology. What instrumentation methodology and metric framework enables reproducible, attributable performance measurement for MCP-mediated DBMSs?

The questions build on one another. RQ1 establishes whether the problem is quantitatively significant, and RQ2 examines its source. RQ3 then tests whether controlled measurements generalize to realistic settings, while RQ4 identifies the measurement infrastructure needed to answer the first three questions reproducibly.

1.5 Thesis Contributions

This thesis makes four integrated contributions, each addressing one or more research questions.

1.5.1 Protocol Tax as a Decomposable Systems Construct

The thesis formalizes the *protocol tax* as a measurable, decomposable quantity rather than an informal observation. The decomposition separates discovery, lifecycle, handover, serialization, parse, transport, and teardown phases from raw database execution time. This taxonomy is implemented as a distributed tracing schema based on the span model introduced by Dapper (Sigelman et al., 2010) and realized through OpenTelemetry (OpenTelemetry Authors, 2025), with database-specific semantic conventions and payload annotations.

The decomposition shows why high end-to-end latency in an agentic DBMS interaction does not necessarily indicate slow SQL execution. As data volume and context demands

increase, middleware conversion and coordination can instead take over the critical path. Identifying the dominant phase and conditions under which it dominates is the first step towards targeted optimization. *(Addresses RQ1 and RQ4.)*

1.5.2 Two-Tier Attribution Framework

Performance differences across MCP DBMS servers do not share one causal structure. Some are under the direct control of server developers, while others arise from inseparable interactions among the DBMS backend, its wire protocol, the runtime environment, and the server implementation. Treating these sources as equivalent can produce misleading conclusions.

The thesis introduces a principled separation into two metric tiers:

- **Implementation Metrics** isolate quantities that depend on server-side engineering choices when the protocol is held fixed. Chapter 3 defines the full set. Some examples include token inflation, payload size at equal semantic content, parse and discovery latency, warm-cache delta, and the boilerplate share of the context window.
- **Coupling Metrics** capture variables arising from inseparable interactions across the DBMS wire format, driver, runtime, and server logic. These include serialization throughput (binary decode plus JSON encode), payload expansion relative to native format, peak memory behaviour, CPU saturation thresholds, connection handover under pooling, and overload back-pressure patterns. These metrics are operationally useful for comparing integrated stacks but cannot be attributed to any single component.

The two tiers report side-by-side. Implementation Metrics give causal diagnostics for engineers writing MCP servers; Coupling Metrics tell practitioners how well a chosen DBMS-MCP stack behaves under representative workloads. Reporting both keeps a single number from being read as the whole answer to either question. *(Addresses RQ2.)*

1.5.3 Two-Track Experimental Design

Benchmarks that include only agents conflate reasoning variance with middleware behaviour. However, ones that exclude agents cannot measure the retry and context dynamics that emerge only under natural-language ambiguity. The thesis therefore uses a two-track methodology to capture both perspectives:

- **Track A (Controlled).** Deterministic harness calls with known-correct SQL. No agent variability. TPC-H scale-factor sweeps provide a single-variable stress axis that isolates middleware sensitivity to data volume while holding query semantics constant (Transaction Processing Performance Council, 2022). Track A produces controlled estimates of protocol and coupling costs without reasoning noise.
- **Track B (Applied).** A real LLM agent workflow under natural-language uncertainty. SQLyzr complexity categories (Abedini, 2026) stratify queries into six structural-complexity categories (C1 to C6) covering aggregation, join depth, nesting, and CASE constructs. Track B measures retry-loop cost, context-window decay, grounding accuracy, and task completion rate. It tests whether implementation-level differences observed in Track A propagate into real interaction outcomes.

Both tracks emit compatible traces under the same instrumentation, which allows their results to be compared directly. An effect observed for a server on a Track A implementation metric can then be tested in Track B; Section 2.7 specifies this linkage. *(Addresses RQ3.)*

1.5.4 MADBench

The thesis produces a concrete benchmark artifact: MADBench. It tests multiple MCP DBMS servers across heterogeneous DBMS backends (PostgreSQL, MySQL, SQL Server, DuckDB) under OLAP-scale payload stress. Results include per-span decomposition, per-metric attribution tier, cross-track validation, and degradation curves under scale-factor sweeps.

MADBench is a coupling benchmark rather than an agent capability benchmark. Its purpose is to determine when protocol tax remains secondary and when it becomes dominant, and to distinguish overhead that can be reduced through implementation changes from overhead inherent to database-protocol integration. *(Addresses all RQs.)*

1.6 A Motivating Example

Consider an analyst who asks an LLM agent:

> *"What are the top 10 customers by revenue in Q4, broken down by country?"*

The agent is connected to a PostgreSQL DBMS through an MCP server. The database contains TPC-H data at scale factor 10. This walkthrough follows six user-visible phases. The formal seven-phase decomposition of Section 1.2 adds Teardown, shown in the accounting table below.

Step 1: Discovery. The agent calls the MCP server's `tools/list` endpoint, and the server returns JSON descriptions of its tools and schema metadata. Depending on the server's design, this response may carry full `CREATE TABLE` definitions, abbreviated column lists, or nothing beyond tool names, consuming hundreds to thousands of tokens before any data-bearing call executes.

Step 2: Schema Reasoning. The agent examines the metadata, identifies the relevant tables (`customer`, `orders`, `lineitem`, `nation`), and determines join paths. If the server omitted foreign-key metadata, it must infer relationships from column names or issue exploratory introspection queries. Schema grounding is a major error source in text-to-SQL systems, with schema-linking errors accounting for 29% to 49% of failures across surveyed methods (Shi et al., 2026).

Step 3: Query Synthesis and Execution. The agent synthesizes a SQL query and sends it through the MCP server. The server translates the JSON-RPC call to a PostgreSQL query, executes it, and receives binary-encoded results over the wire protocol. The DBMS engine finishes in approximately **one second**.

Step 4: Serialization. The server converts the binary result set to JSON text, wraps it in an MCP response envelope, and returns it. For this 10-row answer the serialization cost is trivial; for a query returning 100,000 rows, a common intermediate result in analytical workflows, serialization time becomes measurable and JSON payload size can expand significantly over the native binary encoding.

Step 5: Retry (Conditional). If the initial query fails because the agent selects incorrect join paths due to missing schema metadata, the entire sequence repeats. This consumes additional tokens, wall-clock time, and context-window budget. The retry is not caused by the agent alone; it also depends on the server's metadata exposure policy.

Step 6: Total Accounting. Table 1.1 summarizes the illustrative cost breakdown.

DBMS execution took roughly one second. The remaining time, potentially seconds to minutes, was protocol tax. This thesis measures the ratio between protocol tax and execution time.

Phase	Illustrative Cost	Category
Discovery	100s to 1000s of tokens	Protocol Tax
Schema reasoning	seconds	Protocol Tax
Query execution	$\sim$1 **s**	DB Engine
Serialization (10 rows)	$<$1 ms	Protocol Tax
Serialization (100K rows)	100s of ms	Protocol Tax
Retry loop (if triggered)	seconds to minutes	Protocol Tax
Teardown	10s to 100s of ms	Protocol Tax

Table 1.1: Illustrative cost breakdown for a single MCP-mediated analytical query. The DBMS engine still accounts for a small fraction of the total interaction time relative to protocol tax.

The same scenario changes when it runs through another MCP server for the same PostgreSQL database. This server uses CSV output instead of JSON, saving roughly 29% of tokens on small result sets (Axiom, 2025). It also exposes foreign-key metadata in its discovery response, eliminating the retry, and maintains a warm connection pool instead of making per-query lifecycle calls. These implementation choices substantially reduce protocol tax even though the query and database engine remain the same; the difference is entirely implementation policy.

Switching the backend from PostgreSQL to DuckDB changes the serialization boundary. DuckDB may run in-process, eliminating the network hop while still requiring columnar-to-JSON conversion. Its binary encoding also differs, as may the runtime environment (Python, Go, or Node.js). Some of these differences are under the MCP developer's control, while others are not. Distinguishing between them is the attribution problem quantified in this thesis.

An organization choosing between two MCP servers for their analytics stack needs to know whether the performance difference because of how the server was built (fixable by the developer) or because of inherent properties of the DBMS-protocol coupling (architectural, requiring a different stack choice)? Current benchmarks cannot answer this question. MADBench can.

1.7 Approach and Methodology Overview

1.7.1 Workload Design

The benchmark uses two complementary workload axes. Track A uses TPC-H (Transaction Processing Performance Council, 2022) to measure the effect of data volume. Its schema and query families remain fixed while the scale factor changes. This keeps query semantics constant, making it easier to determine whether increases in protocol overhead come from the middleware or from database execution. Track B uses SQLyzr (Abedini, 2026) to measure query complexity. SQLyzr groups queries into six structural categories (C1 to C6) based on features such as aggregation, join depth, nesting, and CASE expressions. We emphasize multi-join and nested queries because schema linking is a leading source of text-to-SQL failures (Shi et al., 2026; Wang et al., 2025; Hong et al., 2026). Separating data volume from query complexity helps identify which layer causes an observed effect. Section 2.6 explains this design in more detail.

1.7.2 Systems Under Test

The benchmark tests multiple MCP DBMS servers across four DBMS backends: PostgreSQL, MySQL, SQL Server, and DuckDB. These backends differ in wire protocols, binary encodings, runtime environments, and connection models (PostgreSQL Global Development Group, 2025; Oracle Corporation, 2025a; Microsoft Corporation, 2026; DuckDB Foundation, 2025).

Servers are selected to span the design space documented in Chapter 2: varying tool counts (single-tool to dozens-of-tools), schema depth (absent to rich foreign-key metadata), connection models (pooled to explicit multi-step lifecycle), output formats (JSON, CSV, vertical text), and safety mechanisms (keyword filtering to AST-level validation to template-only architectures).

1.7.3 Instrumentation

All measurements use OpenTelemetry-based distributed tracing (OpenTelemetry Authors, 2025; Sigelman et al., 2010). We record each phase of an MCP interaction as a separate span and attach metadata such as payload size, row count, and streaming or buffering mode. This makes it possible to identify bottlenecks that would otherwise be hidden in the tool call duration. We report latency as percentile distributions rather than averages

because slow requests can strongly affect user experience under load.(Dean and Barroso, 2013). Chapter 4 defines the spans recorded for each phase, explains how we verify that tracing does not significantly affect the measurements, and describes the rules used to exclude invalid results.

1.7.4 Analysis Framework

Results are reported using both attribution tiers. Implementation Metrics compare servers on the same backend to show how server choice affects performance. Coupling Metrics compare backends to show overhead caused by the integration. We then compare the results from Track A and Track B to see whether the overhead measured without an agent also affects agent performance. We test whether higher token use, parsing cost, or serialization overhead is linked to more retries, worse context-window performance, or lower task completion. Finally, the TPC-H scale sweeps show when protocol overhead changes from a small cost to the main source of overhead. This transition is the main empirical finding of the thesis.

1.8 Scope and Limitations

1.8.1 In Scope

- OLAP-style analytical queries over relational DBMSs.
- MCP as the agent-to-DBMS protocol.
- Four DBMS backends: PostgreSQL, MySQL, SQL Server, DuckDB.
- Single-agent workflows with iterative tool use.
- English natural-language queries for Track B.
- Protocol-level decomposition and attribution under controlled scale sweeps.

1.8.2 Out of Scope

- OLTP transactional workloads with low result cardinalities, where protocol tax is likely subdominant.

- Non-MCP agent protocols (A2A, ANP, custom REST integrations).
- Non-relational backends (graph DBMSs, document stores, vector DBMSs).
- Multi-agent collaborative workflows where inter-agent coordination adds a separate overhead layer.
- Prompt engineering optimization. This thesis evaluates the interaction pipeline rather than the model, and prompt strategy is held constant across comparisons.

1.8.3 Attribution Boundaries

The two-tier attribution model is directional, not absolute. Some implementation-level interventions (schema caching, response-shape compression) can indirectly influence coupling behaviour, and coupling-level constraints can motivate implementation adaptations in return. The tiers point to the layer that most likely drives an effect under controlled conditions, while acknowledging that the layers still interact.

If protocol tax dominates at OLAP scales, that does not imply equal dominance in low-volume transactional queries or short tool exchanges. The contribution lies in identifying where dominance emerges and why, not in asserting universal behaviour. A concurrent industrial benchmark reports sub-millisecond MCP protocol overhead on point-read workloads across PostgreSQL, MySQL, SQL Server, and SQLite (Faucet, 2026), consistent with the scoping above: at low result cardinality the protocol layer is subdominant, and this thesis targets the OLAP setting where result-set growth and schema depth make it dominant.

1.9 Thesis Organization

The remainder of this thesis is organized as follows.

Chapter 2 (Literature Review) surveys the research threads that converge on the protocol-tax problem: DBMS-LLM integration and the shift from data shipping to knowledge shipping, MCP architecture and ecosystem heterogeneity, MCP and agent benchmarks (including trace-first attribution work), schema grounding as a performance bottleneck, workload design principles for middleware benchmarking, and instrumentation methodology for multi-component agentic pipelines. The chapter ends with a formal research gap statement.

Chapter 3 (MADBench: Workload, Dataset, and Metrics) specifies the benchmark. It defines the two-track workload (TPC-H scale sweeps for Track A, SQLyzr C1 to C6 stratification for Track B), the seven-phase decomposition of Protocol Tax and its nine-term accounting identity, the Implementation Metric family and the Coupling Metric family, and the three formal properties (completeness, exclusivity, observability) that measurements must satisfy.

Chapter 4 (Benchmarking Framework) describes the instrumentation and execution harness: the OpenTelemetry span contract, the calibration and sampling protocols, the systems-under-test matrix (MCP servers $\times$ DBMS backends), and the Track A and Track B execution loops that emit compatible traces.

Chapter 5 (Experiments and Results) presents the empirical findings. It reports Track A per-span decomposition and degradation curves under scale-factor sweeps, Track B retry-loop cost, context-window decay, grounding accuracy, and task completion stratified by SQLyzr categories, and the cross-track analysis that tests whether Track A measurements predict Track B outcomes.

Chapter 6 (Conclusion and Future Work) synthesizes findings across both tracks and both metric tiers, discusses implications for MCP server design and DBMS-MCP stack selection, addresses threats to validity, and lays out the deferred-work agenda: Property 3.1 calibration via a splittable T_{teardown}, a thesis-pinned LLM ablation, full C1 to C6 category coverage, expansion of the Track B paraphrase count from one to three, an n=6 server expansion for RQ3 statistical power, and Track B at scale factor 10.

Chapter 2

Literature Review

2.1 Middleware Friction in Agentic Data Systems

DBMS performance evaluation has traditionally treated the interface between client and DBMS as a stable systems layer. It is characterized by mature drivers, compact binary encodings, and predictable request-response patterns. That assumption weakens when a large language model, rather than a deterministic application, becomes the primary query interface. An LLM client needs more than tuples. It also needs schema descriptions and contextual information represented as text during reasoning (Akillioglu et al., 2025; Shi et al., 2026). This shifts the focus from data shipping toward knowledge shipping, where end-to-end latency may increasingly include context assembly, serialization, and repeated interaction overhead in addition to raw DBMS execution(Liu et al., 2026; Akillioglu et al., 2025; Shi et al., 2026).

The Model Context Protocol (MCP) sits at this boundary. It standardizes model-client-server interactions through JSON-RPC messages, tool discovery, and schema-driven invocation contracts (Anthropic, 2025; JSON-RPC Working Group, 2010). This provides a common interface across heterogeneous agent systems. Standardization reduces integration friction, but it also introduces a performance surface unlike native DBMS protocols built around typed binary transport and cursor semantics (PostgreSQL Global Development Group, 2025; Oracle Corporation, 2025a; Microsoft Corporation, 2026). Protocol attribution has also been reported in the broader agent literature. AI-NativeBench reports that inference dominates end-to-end latency in general agentic workloads, with protocol overhead contributing a smaller share (Wang et al., 2026). Other MCP benchmarks evaluate long-context reasoning, multi-step tasks, and security-related behaviour (Luo et al., 2025a; Guo

et al., 2026; Yang et al., 2025b). Whether these observations persist in OLAP-scale DBMS couplings, where larger payloads and repeated tool interactions may move protocol overhead onto the critical path, is the central question addressed in this thesis.

This thesis introduces the concept of the protocol tax, which includes all latency and resource consumption in an agentic DBMS interaction except raw DBMS execution time: the combined effect of discovery, lifecycle management, handover, serialization, parsing, network transport, and teardown. The motivating hypothesis is that for OLAP workloads at TPC-H scale factor 1 and above, protocol overhead may dominate execution latency by orders of magnitude. The hypothesis is based on two observations: modern DBMS execute analytical queries efficiently, while LLM-mediated processing can become a major end-to-end bottleneck (Transaction Processing Performance Council, 2022; Akillioglu et al., 2025).

The remainder of this chapter builds the argument in seven steps. Section 2.2 traces the shift from data shipping to knowledge shipping and locates where end-to-end latency accumulates outside the query engine. Section 2.3 examines MCP architecture and how ecosystem heterogeneity confounds cross-server comparison. Section 2.4 reviews MCP and agent benchmarks, isolates the diagnostic gap for DBMS couplings, and positions AI-NativeBench's trace-first evidence against the OLAP setting this thesis targets. Section 2.5 links metadata quality and schema-grounding failures to implementation-level decisions in MCP servers. Section 2.6 justifies the workload design, including TPC-H and SQLyzr as orthogonal stress axes. Section 2.7 synthesizes distributed-tracing practice into a two-track, two-tier attribution methodology. Section 2.8 consolidates the research gap and states the contributions that follow. Across these areas, prior work identifies symptoms of middleware friction but does not provide a measurement framework that attributes them to a cause.

2.2 From Data Shipping to Knowledge Shipping

2.2.1 Classical Data Movement and Interface Stability

The dominant design objective in classical DBMS connectivity has been efficient data movement between storage engines and application runtimes. Starting with early relational engine design, the field emphasized data independence, structured schemas, and stable query semantics (Codd, 1970; Stonebraker and Rowe, 1986). Over time, this objective produced mature interfaces such as JDBC and DB-API, with explicit connection/cursor models and type-aware result metadata (Oracle Corporation, 2026; Lemburg, 1999). At

the protocol layer, mainstream systems formalize connection lifecycle, command phases, message framing, and cursor-style fetch behaviour (PostgreSQL Global Development Group, 2025; Oracle Corporation, 2025a). In this environment, benchmark practice has traditionally centered on execution-time and throughput behaviour in the DBMS core (Transaction Processing Performance Council, 2022).

2.2.2 LLM Agents as Query Producers

This assumption no longer holds when the query producer is an LLM agent. The model must infer schema semantics, synthesize joins, and maintain conversational state across iterative tool calls, so the interface now carries explanatory context in addition to tuples: schema descriptions, guidance for later reasoning turns, and planning metadata (Akillioglu et al., 2025; Shi et al., 2026). Query output generation no longer dominates; in multi-step settings, context growth and protocol text consume a meaningful portion of the interaction budget (Shi et al., 2026; Luo et al., 2025a).

Recent DBMS-systems commentary describes a shift toward agent-first access based on iterative model-tool interaction rather than single-shot client requests (Liu et al., 2026). Liu et al. (2026) use the term *agentic speculation* for the resulting pattern of repeated tool calls, metadata lookups, and refinement. This type of workload differs from transactional or dashboard-style access in its scale, heterogeneity, and redundancy.

Akillioglu et al. (2025) report that processing about 17,000 rows took roughly five hours on a single NVIDIA A100 GPU with a locally served Llama 3.1 (8B) model. In this text-heavy and iterative setting, engine execution remained efficient, but middleware and inference hindered end-to-end execution time.

Fernandez et al. (2023) argue that LLMs will shift data-management design from query execution toward model-facing interfaces and orchestration. Similarly, Liu et al. (2025) show that inference throughput and cost can dominate relational analytics pipelines. They report that row and field reordering for KV-cache reuse improves job completion time by up to 3.4× while reducing API cost by 32%. These results show that call ordering and batching are primary optimization concerns in LLM-mediated analytics.

2.2.3 Knowledge Shipping as a Third Mode

Distributed systems traditionally distinguish moving data to compute from moving compute to data. LLM-mediated querying introduces a third mode: knowledge shipping. In this

mode, systems repeatedly provide schema and semantic context as text for the model to use during an interaction (Akillioglu et al., 2025; Shi et al., 2026). This context tends to be verbose, repetitive, and designed for interoperability rather than throughput.

Modern analytical engines have spent years reducing this class of overhead within the data layer. Columnar and vectorized execution emphasize cache efficiency and low materialization cost (Boncz et al., 2005; Melnik et al., 2010; Kersten et al., 2018). Kersten et al. (2018) report single-threaded TPC-H scale factor 1 runtimes on production engines ranging from roughly ten milliseconds for Q6 to a few hundred milliseconds for Q18. MCP-mediated paths add text-based messaging and repeated serialization and parsing boundaries between engine output and model input, although the magnitude depends on server design and workload shape (Anthropic, 2025; Luo et al., 2025a; Wang et al., 2026). These costs shift where end-to-end time accumulates without negating engine efficiency.

2.2.4 Schema Linking as a Dominant Failure Source

Text-to-SQL surveys and error analyses rank schema understanding and schema linking among the top failure sources, especially with broad schemas, ambiguous names, and variation across DBMS(Shi et al., 2026; Hong et al., 2025; Wang et al., 2025; Hong et al., 2026). Metadata quality therefore affects both correctness and efficiency. Sparse metadata can lead to incorrect joins, missing predicates, and retries, while overly verbose metadata increases token and parsing costs. The interface must balance semantic completeness with transport efficiency.

2.2.5 The Missing Metric Framework

DBMS-LLM integration therefore cannot be evaluated with legacy notions of query latency or throughput alone. Fast in-engine execution can coexist with slow user-visible response when schema retrieval, tool negotiation, serialization, and repeated reasoning loops dominate end-to-end behaviour (Akillioglu et al., 2025; Liu et al., 2026; Luo et al., 2025a).

Existing work lacks a metric framework that decomposes this middleware behaviour into attributable components. It identifies the challenge and proposes architectural responses but rarely provides reproducible phase-level accounting across discovery, lifecycle, handover, serialization, parsing, transport, and teardown. It also does not distinguish protocol-design costs from implementation choices in specific MCP servers. This attribution gap motivates the two metric tiers developed in this thesis: Implementation Metrics and Coupling Metrics.

2.3 MCP Architecture and Ecosystem Heterogeneity

MCP introduces a host-client-server pattern in which models invoke tools through standardized message envelopes, capability discovery, and schema-described arguments. The architecture separates model providers from tool providers: hosts manage conversations, clients mediate requests, and servers expose tools and the metadata needed to invoke them (Anthropic, 2025). This common interface allows model-facing products to use different tools without a custom adapter for each backend.

MCP uses JSON-RPC at its data layer, and common implementations use text-oriented transports such as stdio and server-streamed channels (Anthropic, 2025; JSON-RPC Working Group, 2010; Luo et al., 2025a). The resulting encoding and transport costs are small for many tools, especially when responses are short and metadata is static. DBMS couplings differ. They can return many rows with typed and repetitive values, and binary-to-text expansion increases transfer and parsing work at OLAP scales (Transaction Processing Performance Council, 2022; PostgreSQL Global Development Group, 2025). AI-NativeBench shows that architecture can move systems between infrastructure-cost profiles and reports costly failures associated with infrastructure-cost inversions (Wang et al., 2026). This result motivates direct measurement of protocol-path costs for large payloads.

Langdale and Lemire (2019) show that highly optimized JSON parsing is fast but remains limited by parser and hardware constraints. Viotti and Kinderkhedia (2022) report substantial size differences across families of JSON-compatible binary formats. Liu et al. (2023) identify read/write and compression tradeoffs across Arrow, Parquet, and ORC conversion paths. Practitioner benchmarks report smaller payloads and faster serialization with Protocol Buffers than with JSON in selected workloads, although the magnitude depends strongly on benchmark design (gRPC Authors, 2016).

Runtime choice adds another source of variation. Talluri et al. (2025) report that implementation pathway and runtime context can substantially change JSON-processing overhead. Public MCP benchmark reports also show cross-language performance differences under shared harness conditions across Java, Go, Node.js, and Python implementations, even when task shape is similar (TM Dev Lab, 2026). Serialization throughput is therefore classified as a Coupling Metric because runtime, parser implementation, and backend data shape jointly determine it.

Adoption is moving MCP from niche integration layer toward common agent infrastructure. There is rapid growth in server directories, framework integrations, and cross-provider support, including uptake by major AI ecosystem participants (Hou et al., 2026). Benchmarks built around MCP-mediated tool use are appearing, implying consolidation around

a shared interface even as evaluation methodologies stay heterogeneous (Luo et al., 2025a; Guo et al., 2026; Fan et al., 2025).

MCP standardizes message exchange but leaves tool semantics, payload design, and lifecycle policy to server developers. It also does not specify which metadata a server must expose, how verbose schemas should be, or whether connection state must appear in the tool interface. This freedom supports broad participation but also creates implementation variance. As a result, two fully compliant servers can have very different latency and token footprints for the same logical task.

The heterogeneity appears in design choices specific to DBMS coupling. Servers may expose a minimal two-tool surface with implicit connection state or require explicit connect-execute-disconnect sequences. They may return compact arrays with positional typing or verbose per-row objects with repeated keys and natural-language descriptors. Discovery output may include foreign-key and relationship metadata or omit it for simplicity or risk reduction. Each decision changes downstream behaviour without affecting MCP compliance (Anthropic, 2025; Hou et al., 2026).

2.3.1 Tool Granularity and Discovery Overhead

In MCP, a tool is a named operation that a server makes available to an agent. Each tool has a description and an input schema that tell the agent what the operation does and which arguments it accepts (Anthropic, 2025). A DBMS MCP server may expose tools to execute SQL, inspect a table, or manage a connection. Tool count affects the discovery payload size and, in turn, how much of the LLM's context is used before query execution.

We compare eight deployed DBMS MCP servers using their public documentation and source repositories. Table 2.1 summarizes their designs. Chapter 4 evaluates three of these servers: the Anthropic PostgreSQL reference server, DBHub, and mcp-alchemy. The surveyed servers expose between one and several dozen tools.

At one extreme, the archived Anthropic PostgreSQL server exposed a single `query` tool. It offloaded schema discovery to MCP Resources that returned column names and types from `information_schema.columns`, resulting in minimal discovery overhead but also minimal agent guidance (Anthropic, 2024). DBHub takes a middle approach with two primary tools (`execute_sql` and `search_objects`) and progressive disclosure. The agent can request object names, summaries, or full detail to control metadata volume per call (Bytebase, 2025). Oracle SQLcl exposes five explicit lifecycle tools (`list-connections`, `connect`, `disconnect`, `run-sql`, `run-sqlcl`); a single user question therefore requires 5 to 8 tool calls to traverse the lifecycle (Oracle Corporation, 2025b; Hartenfeller, 2025).

SqlAugur exposes six core tools at startup and dozens more through a dynamic toolset-loading mechanism, allowing the agent to enable capabilities as needed rather than parsing the full surface in advance (Bentham, 2024).

Google's MCP Toolbox for DBMSs is the outlier: it eliminates LLM-generated SQL entirely (Google Cloud, 2025). A developer defines parameterized SQL templates in YAML; the LLM sees only tool descriptions and fills parameter values, never encountering raw schema. Across 40+ supported DBMS sources, the Toolbox ships 28 prebuilt tools for PostgreSQL alone, roughly 19,000 tokens when fully loaded; selective loading reduces a single-tool definition to 579 tokens, a 33× reduction (Google Cloud, 2025).

Ecosystem-wide data confirm that most servers expose relatively few tools. Bloomberry (2026) analyzed over 1,400 MCP servers and found a mean of 13.4 tools, a median of 5, and nearly half exposing 4 or fewer. DBMS servers generally fall at the lower end of this distribution.

Table 2.1: DBMS-facing MCP server design axes (public documentation snapshot, March 2026).

Server	Tools	Schema	Connection	Format	Safety
Anthropic PG (archived)	1 + Resources	Basic	Preconfigured pool	JSON	Read-only tx
Oracle SQLcl	5 lifecycle	Data dictionary	Explicit lifecycle	CSV	Restrict levels
MySQL (benborla)	1	None	Pooled, implicit	JSON	Parser + rate limit
DuckDB (ktanaka)	1	None	Local file	Text	Read-only flag
SqlAugur	core + dozens dynamic	Rich (FKs, ER)	Session-oriented	JSON	AST validation
DBHub	2 + custom	Progressive	Gateway, multi-DB	JSON	Keyword filter
MCP Toolbox (Google)	28 prebuilt	Hidden	Centralized pool	JSON	Prepared stmts
MotherDuck DuckDB	5 catalog	Catalog tools	Multi-source	JSON (1024-row cap)	Read-only flag

The benchmark measures the effect of tool count through discovery latency and context consumption. Discovery overhead records the tokens and milliseconds used before the first data-bearing call.

2.3.2 Schema Depth, Metadata Richness, and Grounding Fidelity

Tool count describes only part of the discovery surface. The schema information exposed by each tool is also important. Some servers provide no dedicated schema information, while others hide the raw schema behind templates. The community MySQL server (Borla, 2024) and standard DuckDB server (ktanaka101, 2024) provide no dedicated schema tools. The agent must instead issue and interpret introspection SQL such as `SHOW TABLES` and `DESCRIBE`. The archived Anthropic PostgreSQL server's Resources return `column_name` and

`data_type` from `information_schema.columns`, but omit nullability, defaults, and foreign keys (Anthropic, 2024). The designcomputer MySQL server separates read and write operations across 5 tools, including a dedicated `describe_table` tool (designcomputer, 2024).

mcp-alchemy provides richer schema information through its `schema_definitions` tool, which returns column names, types, primary keys, foreign keys, and nullable flags. Within the surveyed population, it is the only MySQL-compatible MCP server that exposes FK metadata through a dedicated tool (Kaagaard, 2025). SqlAugur provides additional detail: `describe_table` returns Markdown-formatted columns, indexes, foreign keys, and constraints, while other tools generate ER diagrams and traverse dependency graphs (Bentham, 2024). At the opposite end, MCP Toolbox hides the schema by construction; the LLM works through parameterized templates and never encounters raw table definitions (Google Cloud, 2025).

Output format also affects metadata cost. mcp-alchemy uses vertical output with one field per line, while most servers emit per-row JSON objects that repeat key names. At scale, this difference can substantially increase token cost (Kaagaard, 2025).

These choices directly affect schema grounding (Section 2.5). Without FK metadata, an agent cannot infer join paths from discovery alone. It must guess, issue exploratory queries, or fail. Each retry adds latency and token cost. Schema depth is therefore an implementation variable with measurable effects on grounding fidelity.

2.3.3 Connection Lifecycle and Serialization Boundaries

Deployed servers use three connection patterns. Under the *preconfigured/implicit* pattern, the server initializes a connection pool at startup and the agent does not issue lifecycle commands. Anthropic PG, DuckDB, and benborla MySQL follow this pattern (Anthropic, 2024; ktanaka101, 2024; Borla, 2024). Under an *explicit lifecycle*, Oracle SQLcl requires `list-connections` → `connect` → `run-sql` → `disconnect`. This produces 5 to 8 MCP round-trips per user question. Before version 25.4, Oracle SQLcl also used a 60-second timeout (Oracle Corporation, 2025b; Hartenfeller, 2025). Under the *gateway* pattern, DBHub mediates multiple backends behind a single MCP surface through DSN configuration (Bytebase, 2025), while MCP Toolbox provides centralized connection pooling across 40+ sources with IAM-based authentication (Google Cloud, 2025).

The number of serialization boundaries differs between embedded and client-server databases. A PostgreSQL MCP server crosses three boundaries: MCP JSON-RPC encoding, the PostgreSQL wire protocol, and PostgreSQL's internal-to-wire conversion. A standard

DuckDB Python server crosses two: MCP JSON-RPC and DuckDB's columnar-to-Python marshaling. The `duckdb_mcp` native extension reduces this count to one by running the MCP handler within the DuckDB process (Sterling, 2025). Both DuckDB paths must still convert results to JSON text, which reduces some of the benefit of in-process execution. MCP also bypasses DuckDB's zero-copy Arrow integration, which is its most efficient output path.

MotherDuck's DuckDB MCP server caps query results at 1,024 rows and 50,000 characters by default (MotherDuck, 2025), an explicit admission that MCP transport is not designed for bulk data transfer.

Result format affects payload volume. Practitioner measurements report that CSV uses approximately 29% fewer tokens than per-row JSON objects, or roughly 14 fewer tokens per row (Axiom, 2025). TSV uses 30% to 40% fewer tokens (Page, 2026). Oracle SQLcl uses CSV, mcp-alchemy uses vertical output, and the `duckdb_mcp` extension supports JSON, JSONL, Markdown, CSV, and plain text (Sterling, 2025).

Connection model determines lifecycle call count, while serialization format determines payload volume. Both are implementation variables that the benchmark's per-span instrumentation can isolate from protocol-level overhead.

2.3.4 Safety Models and the Security-Performance Boundary

Safety mechanisms range from absent to architecturally enforced, and each level carries a distinct performance cost.

The archived Anthropic PostgreSQL server attempted read-only enforcement through `BEGIN TRANSACTION READ ONLY`. However, the `pg` client library accepts semicolon-delimited multi-statements, so a manipulated agent could issue `COMMIT; DROP SCHEMA public CASCADE;` in a single tool call, as documented by Datadog Security Labs (Datadog Security Labs, 2025). The Anthropic SQLite reference server contained a classic SQL injection vulnerability caused by unsanitized string concatenation, which Anthropic declined to fix, citing the server's archived status (Trend Micro Research, 2025). Supabase MCP was vulnerable to prompt injection leading to full DBMS exfiltration. Its `service_role` key bypassed row-level security, and an attacker could place exfiltration instructions in a support ticket that the agent would read and execute (General Analysis, 2025). Xata MCP shared Anthropic's `pg`-package vulnerability, with semicolon-delimited multi-query strings bypassing SELECT-only validation (nodejs-security.com, 2025).

Four documented SQL injection vulnerabilities across four independent MCP DBMS servers, all from published security research, show a consistent pattern: application-level

SQL validation (read-only transactions, keyword checking, query-prefix inspection) is broken across implementations. DBHub checks whether queries start with `SELECT`, a heuristic that is trivially circumvented (Bytebase, 2025).

Other servers use stronger defenses. SqlAugur uses Microsoft's TSql170Parser for AST-level validation, blocking stacked queries, comment injection, and encoding bypasses at the syntax level, and enforces rate limiting via a token-bucket algorithm (`MaxQueriesPerMinute=60`, `MaxConcurrentQueries=5`) (Bentham, 2024). MCP Toolbox eliminates SQL construction through prepared statements with bound parameters and supports OIDC-authenticated parameters from ID token claims, so the LLM never constructs or sees SQL (Google Cloud, 2025). PlanetScale creates and destroys ephemeral credentials per query, routes reads to replicas, requires human confirmation for DDL, and blocks `UPDATE`/`DELETE` without `WHERE` and `TRUNCATE` (PlanetScale, 2025).

Security mechanisms also affect performance. AST validation adds parse latency per query, ephemeral credentials add round-trips, and metadata restrictions intended to prevent exfiltration can increase grounding errors and retries that inflate total latency. Eliminating SQL construction avoids injection entirely but requires templates to be authored in advance and prevents the agent from composing novel queries. The benchmark instruments these tradeoffs through validation latency in per-tool spans, credential overhead in lifecycle call counts, and grounding degradation in retry rates and schema-linking accuracy.

These four design axes vary widely even though all servers use MCP. Tool counts range from one to several dozen. Schema exposure ranges from absent to architecturally hidden, and connection lifecycles require between zero and eight round-trips. Result formats produce different payload sizes; for example, CSV uses roughly 29% fewer tokens than per-row JSON objects. Safety mechanisms range from bypassable string checks to complete SQL elimination. MCP defines message framing but does not define metadata volume, lifecycle design, payload shape, or validation strategy (Anthropic, 2025; Hou et al., 2026). The benchmark measures these differences through discovery overhead, schema-grounding fidelity, lifecycle call count, serialization cost, and safety-induced latency under controlled workload pressure.

Native DBMS protocols evolved toward constrained semantics for connection lifecycle, type transmission, and cursor control because their target is high-throughput data exchange. PostgreSQL's frontend/backend protocol, the MySQL 8.4 client-server manual, and Microsoft's MS-TDS specification encode type and framing decisions tuned for DBMS interaction rather than generic tool ecosystems (PostgreSQL Global Development Group, 2025; Oracle Corporation, 2025a; Microsoft Corporation, 2026). When MCP wraps these systems, transport generality and backend specialization meet at a boundary where

conversion overhead and policy mismatch surface.

The four documented injection vulnerabilities above, reported by Datadog Security Labs, Trend Micro, General Analysis, and nodejs-security.com, share one failure mode: application-level SQL validation through read-only transactions, keyword-prefix checking, or query-string inspection is bypassable (Datadog Security Labs, 2025; Trend Micro Research, 2025; General Analysis, 2025; nodejs-security.com, 2025). MCP-SafetyBench and MCPSecBench confirm that tool misuse, prompt injection, and schema exfiltration stay active when servers expose execution primitives to autonomous agents (Yang et al., 2025b; Zong et al., 2026). Stronger defenses (AST-level parsing, prepared-statement-only architectures, ephemeral credentials) improve resistance but each adds measurable overhead in parse latency, round-trip count, or grounding-degrading metadata restriction (Bentham, 2024; Google Cloud, 2025; PlanetScale, 2025).

Together, security and performance pressures make metadata richness a controlled variable, not a fixed protocol property. Compressing schema to limit attack surface or context cost can eliminate join-path information and produce invalid SQL; exposing extensive descriptive metadata for grounding clarity inflates token budgets and parse costs. The cheapest validation strategies are also the least effective, while the most reliable carry the highest per-query overhead. Neither outcome follows from MCP specification text alone; both are implementation policy choices with measurable consequences (Anthropic, 2025; Hou et al., 2026; Akillioglu et al., 2025).

Context-aware MCP extensions have been proposed to mitigate stateless request-response operation in multi-step workflows. CA-MCP introduces a shared context substrate so collaborating servers reuse structured state across turns, reducing repeated transfer and improving coordination in complex tasks (Jayanti and Han, 2026). Their emergence suggests baseline stateless interaction imposes avoidable lifecycle and context costs for workflows that resemble DBMS sessions rather than isolated function calls (Anthropic, 2025; Jayanti and Han, 2026).

This has a direct measurement consequence. If baseline and context-aware variants produce different discovery and lifecycle behaviour, conclusions depend on which variant and server policy are tested. A single end-to-end latency number is under-specified without an explicit account of discovery cadence, context-reuse policy, and payload-shaping rules; without those controls, differences across couplings can be misread as protocol effects when they are implementation effects.

The gap can be stated precisely. AI-NativeBench provides rigorous trace-first decomposition for general agentic workloads, but, to our knowledge, existing MCP studies rarely report DBMS-specific decomposition under controlled high-volume payload sweeps (Wang

et al., 2026; Luo et al., 2025a; Guo et al., 2026). Treating MCP as a generic interoperability layer and scoring task outcomes serves capability assessment but not DBMS middleware attribution. What remains unaddressed is a framework that separates universal protocol-tax components from implementation-induced variation under high-volume DBMS workloads.

2.4 MCP and Agent Benchmarks

2.4.1 Task Completion Benchmarks and Their Diagnostic Limits

MCP benchmarking has accelerated, but its measurement targets remain unclear. Much of the literature evaluates whether an agent can complete realistic tasks through MCP servers in domains such as code repositories, productivity tools, and web applications (Luo et al., 2025a; Guo et al., 2026; Mo et al., 2025). For example, MCP-AgentBench includes 33 servers, 188 tools, and 600 queries across six categories. It scores end-to-end task completion rather than intermediate tool-call correctness (Guo et al., 2026). These benchmarks reveal practical failure rates, capability gaps, and safety constraints under realistic orchestration. However, task completion alone provides limited diagnostic value for middleware optimization because it combines reasoning quality, tool semantics, protocol overhead, and backend behaviour.

Task-level evaluations illustrate the limitation. MCP-Universe and related suites show that strong frontier models still underperform on complex multi-step tool tasks, with aggregate success rates far from saturation (Luo et al., 2025a). LiveMCPBench finds that retrieval errors become the dominant failure mode as tool pools scale (Mo et al., 2025). These results identify bottlenecks but do not show whether they arise from planner weakness, weak schema grounding, high discovery overhead, or payload inflation from tool responses.

Several benchmarks add efficiency-oriented indicators. They include tool-call success variants, parameter-use correctness, invocation rounds, and resource-aware metrics. MCP-ToolBench++ and MCP-RADAR extend binary success with richer diagnostic categories and evaluate robustness across diverse tool-use settings (Fan et al., 2025; Gao et al., 2025). ETOM contributes staged orchestration scenarios that expose fragility as workflow complexity increases (Dong et al., 2026). Although these additions improve granularity, their measurements remain largely agent-facing. Serialization spans, parse overhead, and lifecycle costs are rarely instrumented as explicit protocol metrics.

Multi-server orchestration benchmarks report discovery and coordination scaling as attached services grow (Mo et al., 2025; Dong et al., 2026). However, reported latency

growth remains attributed at the scenario level and is not decomposed into protocol exchange time versus server-implementation characteristics such as tool-count policy or metadata verbosity. These results identify scaling behaviour but do not localize its source.

2.4.2 Security Benchmarks and Exploit Attribution

Security benchmarks examine a different set of failures. MCPSecBench and MCP-SafetyBench evaluate injection pathways, tool-poisoning attacks, and policy-bypass scenarios when defenses are insufficient (Yang et al., 2025b; Zong et al., 2026). MCPSecBench reports core vulnerability classes across major platform ecosystems (Yang et al., 2025b). MCP-SafetyBench identifies the MCP server surface as the dominant attack vector in its corpus, accounting for 74.69% of documented attacks. It also reports per-domain attack success rates from roughly 30% for Web Search to 47% for Financial Analysis (Zong et al., 2026). These studies measure exploit success, failure-to-block frequency, and defense efficacy under adversarial prompts. However, the measurements remain at the end-to-end behaviour level. Security hardening may increase or reduce latency depending on validation depth and metadata minimization, but these tradeoffs are rarely decomposed into protocol costs.

Benchmark families now cover capability, robustness, safety, and in some cases trace-level attribution across diverse agent applications (Wang et al., 2026). MCP-mediated agent systems are measurable, fragile in identifiable ways, and sensitive to tool-interface quality. The open area is coupling-centric measurement for high-volume DBMS interactions, where payload size and serialization policy can dominate behaviour that generic task benchmarks do not isolate. Practitioner reporting fills the gap from a different angle: an industrial MCP-DBMS benchmark across PostgreSQL, MySQL, SQL Server, and SQLite reports order-of-magnitude latency variation attributable to indexing and workload shape rather than protocol semantics alone (Faucet, 2026). That work is complementary, targeting deployment ranking rather than protocol-tax decomposition.

2.4.3 The Attribution Blind Spot in DBMS Couplings

The attribution gap has two layers. First, protocol-level instrumentation is uneven rather than absent. AI-NativeBench demonstrates component-level attribution in general agentic systems, but MCP suites do not consistently report serialization throughput, payload expansion, client parse time, or connection-handover overhead under controlled data-volume sweeps (Wang et al., 2026; Luo et al., 2025a; Fan et al., 2025). Second, attribution models remain underdeveloped for DBMS couplings. Even when latency is reported, analysis rarely

distinguishes implementation choices under the same protocol from inherent behaviour of the integrated DBMS-runtime stack.

This distinction matters for the DBMS domain. If one server returns verbose row objects and another compact arrays, parse and token costs diverge although both are protocol-compliant. If one requires explicit connection lifecycle calls and another maintains warm pooled sessions, end-to-end latency changes without any change to SQL semantics or query plans. Labeling both as "MCP overhead" obscures the optimization target: one case is developer-policy dependent, the other coupling dependent.

Comparative studies also face a denominator problem. End-to-end response time aggregates agent inference, protocol exchanges, network transit, backend execution, and post-processing. Comparing this aggregate across models or servers informs users but is weak for systems science: a slower total may reflect stronger but longer reasoning rather than poorer coupling, and a faster total may hide lower answer quality or aggressive metadata omission that raises silent-failure risk. Without decomposition, speed-quality tradeoffs are easy to misinterpret.

2.4.4 Toward DBMS-Style Control for MCP Measurement

DBMS benchmarking provides a useful method for controlling experiments. TPC-style benchmarks use fixed workload definitions, precise metric semantics, and clear reporting rules so that system comparisons remain interpretable (Transaction Processing Performance Council, 2022). Current MCP benchmarks often vary workloads, schemas, tool sets, and agent strategies in the same experiment. This creates realistic scenarios but makes causal attribution difficult because many variables change together.

The benchmark addresses this problem by applying DBMS-style control to middleware measurement while retaining agent behaviour where needed. Its two tracks separate deterministic measurements without an agent from validation with an agent. Track A measures protocol and implementation behaviour under fixed SQL and controlled payload scales. Track B tests whether these differences affect real agent workflows.

Despite this coverage, explicit implementation-versus-coupling attribution stays uncommon in MCP DBMS studies, which often compare "servers" or "agents" as monolithic entities. The Implementation Metrics versus Coupling Metrics split addresses this gap by identifying which observed differences tie to developer choices under fixed protocol rules and which remain entangled with backend wire format, runtime behaviour, and integrated stack characteristics.

This matters for practice, not only for method. Practitioners selecting a DBMS-MCP stack need coupling-level rankings to estimate deployment behaviour; developers improving a server need implementation-level diagnostics to know where interventions matter. A benchmark that reports only one tier serves one of these audiences poorly and the other not at all, so both tiers are needed.

2.5 Schema Grounding and Metadata Quality

2.5.1 Schema Linking as the Primary Bottleneck

Agent failure in text-to-SQL is often framed as model reasoning weakness, but the evidence points elsewhere: schema grounding quality is frequently the primary bottleneck (Wang et al., 2025; Shi et al., 2026). When agents miss the correct tables, columns, and join paths, downstream SQL synthesis cannot recover regardless of decoding strategy (Wang et al., 2025; Shi et al., 2026). Across benchmark families and architectural styles, improvements in retrieval and linking often yield larger gains than decoder-only complexity under weak schema signals (Pourreza and Rafiei, 2023; Shi et al., 2026).

LinkAlign formalizes this in large-scale multi-DBMS settings, where schema-linking complexity grows with DBMS count, naming ambiguity, and heterogeneous domain vocabularies (Wang et al., 2025). Its error analysis reports "Error 1: Target DBMS without Retrieval" at 23.6% of observed failures, making metadata quality both a model-side and a DBMS-retrieval exposure issue, and attributes its gains mainly to better alignment between natural-language intent and schema candidates rather than to changes in SQL generation grammar (Wang et al., 2025). This has consequences for MCP-mediated systems: tool responses that under-specify schema semantics cap achievable quality even with strong base models.

There are studies that reinforce this observation. Shi et al. (2026) summarize error analyses in which schema-linking categories show the highest average error rates (29% to 49%), against roughly 21% to 26% for join-related categories, making grounding quality one of the most productive intervention points for text-to-SQL reliability.

Pourreza and Rafiei (2023) decompose text-to-SQL into schema linking, query classification and decomposition, SQL generation, and self-correction, reporting execution-accuracy gains over few-shot baselines on Spider, with gains persisting under larger and noisier BIRD settings even as absolute difficulty rises (Pourreza and Rafiei, 2023; Li et al., 2023).

Schema representation quality in MCP responses is therefore an upstream determinant of end-to-end query correctness.

2.5.2 Error Taxonomies and Generation-Side Methods

Not all incorrect SQL is equally visible. ErrorLLM proposes a fine-grained SQL-error taxonomy emphasizing attribute-level failures such as attribute mismatch and redundancy, and because agents comply closely with error-guided instructions, a false error signal is costly, so detection must balance precision and recall (Hong et al., 2026). Its taxonomy separates grounding-related errors (selecting nonexistent or semantically mismatched attributes or tables) from composition errors where valid elements are combined incorrectly; both matter, but grounding-related errors are shaped directly by metadata availability (Hong et al., 2026). For MCP-mediated querying this favors verified schema context over merely verbose context.

Generation-side methods such as CHASE-SQL, Gen-SQL, SQLGovernor, and MARS-SQL report robustness gains through candidate diversification, pseudo-schema bridging, fragment-wise constraints, and multi-agent coordination (Pourreza et al., 2025; Shi et al., 2025; Jiang et al., 2025; Yang et al., 2025a). They reduce failure rates on difficult questions but still rely on sufficiently informative schema context: if relationship metadata is absent, candidate expansion can amplify wrong-join hypotheses rather than correct them.

2.5.3 Metadata Quality as a Systems Variable

Metadata quality must be treated as a systems variable in MCP benchmarking (Wang et al., 2025; Shi et al., 2026). In an MCP-mediated workflow, the agent's visible schema is not the DBMS catalog but the transformed representation the server emits, which may omit constraints, compress descriptions, reorder information, or add explanatory text. Grounding quality is therefore partly a server implementation property, not only a model property.

This distinction affects how retries are attributed. If a server exposes tables and columns but omits foreign keys, the agent may repeatedly synthesize invalid joins and re-issue queries until it infers enough implicit signals. The delay looks like model inefficiency, yet causal responsibility is shared with metadata exposure policy. The thesis measures this with an explicit retry-loop cost metric, treating retries as implementation-attributable when incomplete or ambiguous tool-provided schema context triggers them.

Context-window decay creates a related problem. MCP messages contain envelopes, argument schemas, and result wrappers that consume tokens before any semantic content

appears (Luo et al., 2025a; Fan et al., 2025). More verbose boilerplate and repeated descriptors reduce the context available for reasoning on each turn, particularly in long tool-use trajectories. As the conversation grows, the interaction between this verbosity and memory limits degrades both quality and latency.

2.5.4 Complexity-Conditioned Effects and Category-Aware Reporting

Survey analyses confirm that schema-complexity effects are not uniform across query classes (Shi et al., 2026). Simple single-table aggregation is relatively robust, while multi-table joins, nested subqueries, and window-function scenarios are far more sensitive to grounding quality (Shi et al., 2026). This motivates using SQLyzr complexity categories for Track B correlation analysis (Abedini, 2026): if retries cluster in high-complexity categories under one MCP implementation but not another, the benchmark can attribute quality differences to metadata policy with more confidence.

Aggregate scores can hide systematic failures on join-heavy queries that are common in analytical workloads. Category-aware reporting identifies where these failures occur, while latency decomposition explains their performance cost. Together, they provide a more complete evaluation of MCP implementations.

Security reinforces the need for fine-grained analysis. Restricting metadata exposure reduces attack surface but can increase grounding errors and retries, while richer metadata improves grounding yet exposes additional exploitation vectors under weak validation (Yang et al., 2025b; Zong et al., 2026). The tradeoff resists generic prescriptions; it requires measurement frameworks that quantify both safety posture and downstream performance under controlled conditions.

The schema-grounding literature and MCP benchmark evidence show that interface design affects agent quality. Even with a standardized protocol, server choices about tool granularity, schema completeness, descriptor verbosity, and error semantics affect grounding fidelity. These choices are Implementation Metrics and can affect both latency and correctness. Existing benchmarks do not measure these effects while controlling for coupling-level confounds.

2.6 Workload Design for Middleware Benchmarking

2.6.1 Workload Control and Attribution Strength

Workload selection determines what a benchmark can claim. When workloads vary too many dimensions at once, attribution weakens even with sophisticated tooling. DBMS benchmarking has long controlled schemas, query templates, and scaling axes to preserve interpretability (Transaction Processing Performance Council, 2022), and SQLStorm argues that traditional TPC-style suites can underrepresent production SQL behaviour and should be complemented with realistic LLM-generated workloads (Schmidt et al., 2025). The principle matters even more for MCP-DBMS coupling studies, whose objective is to isolate middleware friction under increasing data and interaction pressure.

TPC-H is a strong controlled baseline: a fixed analytical query suite, a stable schema, and a scale-factor parameter that grows data volume while preserving workload structure (Transaction Processing Performance Council, 2022). This yields a single-variable stress axis, the same logical query families and schema semantics with larger result sets and scan footprints as scale grows, so observed changes map more confidently to transport and serialization behaviour than to shifting query semantics.

Most modern text-to-SQL benchmarks, by contrast, target reasoning diversity rather than transport attribution (Shi et al., 2026; Yu et al., 2018; Lei et al., 2025). Spider and Spider 2.0 vary schema domains, dialects, and task complexity, which suits model-generalization studies but complicates middleware causality (Yu et al., 2018; Lei et al., 2025), and SQLStorm adds broad query diversity and LLM-era stress scenarios because LLM-generated workloads surface real SQL functionality that handcrafted suites undercover (Schmidt et al., 2025). These resources stay relevant; they are simply not ideal as sole controlled axes for coupling decomposition.

2.6.2 Payload Sensitivity and the TPC-H and SQLyzr Pairing

Serialization time, parse time, and payload expansion depend on output size and schema message length rather than on abstract query "difficulty". A simple query returning millions of rows can stress middleware far more than a syntactically complex query returning ten rows (Wang et al., 2026; Luo et al., 2025a; Langdale and Lemire, 2019; Viotti and Kinderkhedia, 2022). A middleware benchmark must therefore control data volume first and treat query complexity as a separate, complementary dimension.

The design pairs TPC-H with SQLyzr as the complexity-layer complement (Abedini, 2026), a taxonomy of natural-language and SQL patterns for category-wise analysis of grounding and retry behaviour. Its emphasis on multi-join and nested-query classes follows prior evidence that metadata completeness materially affects schema linking and error recovery (Shi et al., 2026; Wang et al., 2025; Hong et al., 2026). The role is orthogonal, not substitutive: TPC-H drives controlled throughput stress in Track A, while SQLyzr categories characterize failure modes in Track B where agent reasoning is present.

LLM-era datasets strengthen the design. Li et al. (2023) introduce BIRD, a large-scale text-to-SQL benchmark spanning many DBMSs and substantial data volume, targeting noisy values, large schemas, and practical grounding difficulty. SQLyzr-style composition drawing from BIRD and related corpora is therefore suitable for exposing complexity-conditioned failures without abandoning controlled throughput sweeps.

Synthetic-data generation also supports controlled scaling experiments. Patki et al. (2016) describe a relational workflow that models table distributions and inter-table dependencies, then generates datasets of different sizes while preserving their statistical properties.

The two-axis design separates throughput from query complexity. This makes it easier to identify the source of observed behaviour. If middleware overhead increases under fixed TPC-H workloads, the cause is the MCP implementation. If retry-loop cost increases only for high-join SQLyzr categories, the cause is metadata quality.

2.6.3 Cross-Backend Comparability and Reporting Conventions

PostgreSQL, MySQL, SQL Server, and DuckDB differ in wire protocols and runtime ecosystems (PostgreSQL Global Development Group, 2025; Oracle Corporation, 2025a; Microsoft Corporation, 2026; DuckDB Foundation, 2025). These differences are unavoidable and, in coupling studies, informative. However, if workload shape also varies across backends, observed protocol overhead cannot be attributed to a single cause. TPC-H addresses this by providing a common schema and workload across all backends, making coupling-induced differences easier to interpret. For this reason, broad open-domain benchmarks such as Spider 2.0 and SQLStorm are excluded from Track A. This decision preserves causal isolation and does not reflect their general quality as benchmarks.

Historical DBMS methodology supports this choice. Benchmark designers routinely separate microbenchmarks, standardized throughput suites, and application traces because each answers different questions, and no single workload maximizes all forms of validity at once (Transaction Processing Performance Council, 2022; Schmidt et al., 2025). The present

design follows that tradition: controlled TPC-H sweeps for decomposition, category-rich agent tasks for ecological validation.

This reasoning is consistent with the thesis hypothesis that protocol tax dominates at OLAP scales. OLAP queries often produce large result sets, and in LLM-mediated systems larger payloads increase serialization and conversion overhead even when modern DBMSs execute queries efficiently (Stonebraker et al., 2005; Boncz et al., 2005; Neumann, 2011; Wang et al., 2026; Luo et al., 2025a). If protocol overhead dominates under these conditions, the bottleneck lies in the middleware rather than the DBMS.

Reporting conventions are equally important. Following distributed-tracing practice, results should report the scale factor, result cardinality, payload size, and whether outputs are streamed or buffered (Sigelman et al., 2010; OpenTelemetry Authors, 2025). Without this information, identical latency values may represent different transport conditions, making comparisons and reproduction more difficult (Transaction Processing Performance Council, 2022; Sigelman et al., 2010; OpenTelemetry Authors, 2025).

TPC-H and SQLyzr play different roles in the benchmark. TPC-H provides controlled workload scaling for Track A, while SQLyzr categorizes query complexity in Track B. Together, they support both controlled measurement and agent-based validation.

2.7 Instrumentation and Measurement Methodology

2.7.1 Distributed Tracing as Methodological Baseline

Measurement in agentic DBMS systems is a distributed-systems problem. A single user request can trigger schema discovery, multiple tool calls, DBMS round-trips, serialization stages, model inference turns, and post-processing before final output. End-to-end latency collapses these spans into one number, useful for reporting user experience but insufficient for diagnosis: without decomposition, optimization risks targeting the wrong layer.

Classical distributed tracing is the baseline for such decomposition. Dapper established request-level span hierarchies for attributing latency across heterogeneous services (Sigelman et al., 2010), and OpenTelemetry carries this into modern practice through standardized APIs and semantic conventions for cross-language trace collection, context propagation, and metrics and log integration (OpenTelemetry Authors, 2025). Both apply directly to MCP-mediated systems, where host, client, server, and DBMS components can each emit spans.

AI-NativeBench extends this approach with a trace-first methodology. It models agent actions as explicit spans and reports component-level behaviour across different architectures, including MCP-style service configurations. It also introduces the Parameter Paradox and Content-Process Divergence, and reports that inference dominates end-to-end latency in general agent workloads (Wang et al., 2026). The study shows that component-level attribution can change system-level conclusions. It also demonstrates the importance of separating model inference from protocol and infrastructure costs.

Latency-token-aware inference frameworks add another angle. Huang et al. (2025) formalize test-time policies that jointly account for quality, wall-clock delay, and token expenditure under iterative reasoning. This maps naturally to MCP-DBMS coupling, where serialization verbosity and schema boilerplate raise both latency and token consumption; measuring one dimension without the other misstates practical efficiency, especially under long context windows and repeated tool calls.

Traditional DBMS benchmarks rarely require multi-layer attribution because the client side is comparatively simple. Measuring execution time and throughput is usually sufficient when protocol overhead is small relative to DBMS execution time (Transaction Processing Performance Council, 2022). Agentic MCP workflows violate this assumption (Wang et al., 2026; Luo et al., 2025a; Guo et al., 2026). As payload size and context requirements increase, discovery latency, lifecycle management, and parsing overhead can match or exceed DBMS execution time. The interaction must therefore be instrumented as a pipeline of spans rather than a single timed operation (Wang et al., 2026).

2.7.2 Two-Track Design and Metric Attribution

The methodology separates controlled and applied tracks with shared instrumentation semantics. Track A uses deterministic harness calls with known-correct SQL to isolate MCP server and coupling behaviour without agent variability; Track B introduces a real agent workflow to measure retry loops and context decay under natural-language uncertainty. Both emit compatible traces, so differences can be compared directly rather than inferred.

This design fills a gap between two benchmarking traditions (Transaction Processing Performance Council, 2022; Wang et al., 2026; Luo et al., 2025a). DBMS benchmarks provide strong workload control and repeatability but exclude autonomous agents (Transaction Processing Performance Council, 2022; Schmidt et al., 2025), while agent benchmarks include realistic reasoning but sacrifice controllability and low-level attribution (Luo et al., 2025a; Guo et al., 2026); AI-NativeBench is a notable exception for general workflows but does not target DBMS-specific coupling spans (Wang et al., 2026). Combining both

tracks under one instrumentation schema retains causal clarity where needed and ecological validity where necessary.

The explicit metric-attribution framework is a key contribution. Implementation Metrics cover variables traceable to developer decisions within MCP servers. These include schema verbosity, payload formatting policy, discovery tool count, lifecycle workflow, cache warmup behaviour, retry inducement through metadata omissions, and context decay from boilerplate. Coupling Metrics cover variables arising from inseparable interactions across DBMS wire format, driver, runtime, and server logic: serialization throughput, payload expansion relative to native format, peak memory, CPU saturation thresholds, connection handover under pooling, and overload back-pressure.

The binary-format confound shows why. Serialization throughput measured across binary decode plus JSON encode depends on backend behaviour: PostgreSQL, MySQL, SQL Server, and DuckDB expose different wire semantics and runtime data-handling paths, so the decode cost entering MCP is not implementation-neutral (PostgreSQL Global Development Group, 2025; Oracle Corporation, 2025a; Microsoft Corporation, 2026; DuckDB Foundation, 2025). Treating this span as an implementation metric would over-attribute responsibility to MCP developers; classifying it as a coupling metric keeps the accounting honest while still yielding useful rankings.

Payload expansion ratio makes the same point. Its numerator depends on server JSON-shape choices, its denominator on native binary encoding outside server control. Reporting a single ratio without an attribution tier can mislead readers into assuming developer control over the whole metric. The two-tier framework signals that some values are actionable for implementers and others primarily comparative for stack selection.

Spans must record data volume as well as interaction count. A call that returns ten values is not equivalent to one that returns ten million, even when the method name is the same. Each span therefore needs payload-size, row-count, and stream-or-buffer mode tags for meaningful normalization (OpenTelemetry Authors, 2025; Wang et al., 2026). Without these tags, aggregate call durations can hide high-volume bottlenecks.

Results should report latency percentiles and, where relevant, normalize values per byte or per row. Averages can hide slow requests that affect users under load. They can also hide saturation thresholds needed for capacity planning (Sigelman et al., 2010; Wang et al., 2026). For deployment guidance, coupling benchmarks must report typical performance and degradation as concurrency and result size increase (Wang et al., 2026; Luo et al., 2025a).

This combined two-track, two-tier method remains rare in database research (Transaction Processing Performance Council, 2022; Wang et al., 2026; Luo et al., 2025a; Guo et al., 2026). AI-NativeBench and related systems use trace-first attribution for general workflows but do

not measure database coupling spans such as binary decoding and JSON serialization at OLAP volumes (Wang et al., 2026). Database benchmarks provide controlled scale sweeps but do not instrument model-facing protocol stages (Transaction Processing Performance Council, 2022; Schmidt et al., 2025). The proposed framework combines these methods to support reproducible and interpretable measurement of agentic data systems.

2.7.3 Operational Requirements for Reliable Attribution

Several operational requirements follow from this framework, each of which can undermine attribution if neglected.

Span definitions must be consistent across implementations. One server may start its serialization span when row fetching begins, while another may start after the rows become in-memory objects. These values are not comparable even if both traces appear complete. The same problem occurs when one server includes connection-pool wait time in a lifecycle span and another excludes it. Cross-coupling benchmarks must therefore publish span contracts with start and end events, required attributes, and exclusion rules alongside the metric values (Sigelman et al., 2010; OpenTelemetry Authors, 2025; Wang et al., 2026).

Tracing must also be calibrated. Context propagation, timestamp collection, and exporter buffering add overhead. This overhead can be significant for short spans (Sigelman et al., 2010; OpenTelemetry Authors, 2025). The methodology should include baseline runs with tracing disabled or reduced and report the difference. This confirms that tracing does not change the conclusions.

Client-side measurement is a blind spot in much of the literature. Published MCP evaluations foreground server-side or aggregate outcomes (Wang et al., 2026; Luo et al., 2025a; Guo et al., 2026), but in DBMS couplings the cost of client parsing and post-processing for large JSON payloads can be substantial and is sensitive to parser strategy, memory-allocator behaviour, and object-materialization policy (Langdale and Lemire, 2019; Viotti and Kinderkhedia, 2022). Ignoring client cost systematically understates Protocol Tax, especially when server-side timing looks efficient because of aggressive buffering. This thesis treats client parse time as an explicit metric, aligning measurement with user-observed latency rather than server-local completion timestamps.

Percentile reporting matters because median latency can stay stable while P95 and P99 degrade sharply under modest concurrency, particularly when connection handover and serialization contend for CPU and memory (Sigelman et al., 2010; Wang et al., 2026). Users experience outliers as failures even when the average looks acceptable, so decomposition should be paired with percentile distributions and saturation curves, especially across

runtime stacks with different garbage-collection profiles or scheduling models (Wang et al., 2026).

Cross-track linkage benefits from pre-registered hypotheses. If Track A shows high token inflation and parse cost for a server, Track B can test whether that server shows steeper context-window decay and higher retry latency under comparable task categories, and pre-registering such linkages reduces post hoc narrative fitting to noisy data. The instrumentation must support drill-down at multiple granularities: aggregate coupling rankings for stack selection, per-span diagnostics for implementation tuning. A benchmark publishing only one view forces readers into over-generalization or over-specialization.

Three additional controls protect measurement integrity. First, the benchmark must verify trace continuity and parent-child relationships between spans before using timing data. This is important because MCP servers use different languages, frameworks, and observability libraries (Luo et al., 2025a; Guo et al., 2026). Second, CPU-frequency pinning, container isolation, randomized run order, and dispersion reporting reduce hardware and deployment noise. This noise may be small relative to a long database execution span but large relative to a short discovery or handover span (Transaction Processing Performance Council, 2022; Wang et al., 2026). Third, sampling must balance measurement detail against tracing overhead. The benchmark uses deterministic sampling for low-volume phases and adaptive capture for high-volume spans. It must also report the sampling rules so that readers can assess bias (Sigelman et al., 2010; OpenTelemetry Authors, 2025).

Reliable coupling evaluation requires clear span boundaries. It requires consistent sampling rules, runtime controls, and causal classifications. Without these controls, results can be difficult to interpret, even for large datasets. With these controls, protocol-tax decomposition provides a reliable basis for analysis. Monitoring measures current system behaviour. Benchmarking explains why the system behaves that way under controlled conditions.

2.8 Summary and Research Gap

Six research threads converge on one unresolved question. First, DBMS-LLM integration studies show the client-relational interface now carries semantic context in addition to tuples, making metadata transfer and interaction structure central to performance and correctness (Akillioglu et al., 2025; Shi et al., 2026). Second, MCP architecture studies document standardization benefits and ecosystem expansion alongside openness-induced heterogeneity and security-pressure tradeoffs (Anthropic, 2025; Hou et al., 2026; Yang

et al., 2025b). Third, MCP benchmark suites show real-world tool use remains fragile even for strong models, while AI-NativeBench adds trace-first attribution and reports inference-dominant latency in general workflows (Luo et al., 2025a; Guo et al., 2026; Fan et al., 2025; Wang et al., 2026). Fourth, schema-grounding work indicates metadata quality is a major predictor of SQL success, especially in complex join settings (Wang et al., 2025; Hong et al., 2026; Pourreza and Rafiei, 2023). Fifth, workload-design research supports controlled axes for attribution, while SQLStorm critiques TPC-style narrowness and argues for supplementing controlled suites with LLM-generated workloads, together supporting TPC-H scale sweeps as one orthogonal stress axis rather than a complete workload model (Transaction Processing Performance Council, 2022; Li et al., 2023; Schmidt et al., 2025). Sixth, instrumentation research connects Dapper's span concepts with OpenTelemetry standards and agent-level trace-first evaluation (Sigelman et al., 2010; OpenTelemetry Authors, 2025; Wang et al., 2026).

The field has identified both symptoms and partial remedies, and now includes explicit attribution evidence for general agentic systems: in broad workflows, AI-NativeBench finds attributed protocol delay typically lower than inference delay (Wang et al., 2026). The unresolved coupling problem is narrower and DBMS-specific: no established benchmark tests whether that relationship persists when MCP mediates OLAP-scale result sets, binary-to-JSON conversion, and DBMS-specific lifecycle behaviour under controlled scale sweeps (Transaction Processing Performance Council, 2022; Schmidt et al., 2025; Wang et al., 2026; Luo et al., 2025a).

This thesis frames protocol cost as a decomposable systems construct. The protocol-tax lens separates discovery, lifecycle, handover, serialization, parse, transport, and teardown from raw execution time, sharpening the point introduced in Section 1.5.1: aggregate latency in agentic DBMS interactions cannot be read as slow SQL, because middleware conversion and coordination can dominate the critical path as data volume and context demands grow. That reading aligns with both DBMS optimization history and recent agentic-workflow observations (Graefe, 1993; Wang et al., 2026; Luo et al., 2025a).

Heterogeneity under shared standards is the other half. MCP compliance does not imply homogeneous performance, because the specification leaves substantial freedom in tool design, schema exposition, connection policy, and result formatting, so measured differences across DBMS-MCP servers can originate from implementation choices rather than protocol semantics. Existing literature reports these differences descriptively; the present work treats them as measurable implementation variables.

This motivates the Implementation Metrics tier. Metrics such as token inflation factor, payload size for equivalent logical results, parse latency, discovery latency, lifecycle overhead,

cold-to-warm deltas, retry-loop cost from metadata incompleteness, and context-window decay tie to identifiable developer decisions under fixed protocol rules. They are actionable because interventions are local and testable: change schema representation, cache policy, tool granularity, or error-payload behaviour, then re-measure under controlled conditions.

Not all observations permit this attribution. Serialization throughput across binary decode and JSON encode depends jointly on backend wire format, driver behaviour, runtime efficiency, and server implementation; payload expansion relative to native binary mixes an implementer-controlled numerator with a backend-controlled denominator; and CPU saturation and peak memory under load couple runtime and data-shape effects not separable without intrusive redesign. This motivates the Coupling Metrics tier.

The two-tier model does not force a choice between strict causality and operational relevance. Implementation Metrics give developers causal diagnostics; Coupling Metrics give practitioners comparative guidance for stack selection. Reporting both prevents over-claiming while still delivering deployment-relevant evidence, a balance often absent in current MCP and text-to-SQL practice, where metrics are either too aggregate for diagnosis or too narrow for system selection (Shi et al., 2026; Luo et al., 2025a; Guo et al., 2026; Wang et al., 2026).

Attribution also depends on track design. As Section 1.5.3 argues, agent-only benchmarks conflate reasoning variance with middleware behaviour while agent-free benchmarks miss retry and context dynamics; the two-track methodology resolves this by combining deterministic harness measurements with agent-present validation under matched instrumentation. Track A produces controlled estimates of protocol and coupling costs without reasoning noise, and Track B verifies that Track A's implementation-level differences propagate into real interaction outcomes, particularly retry-loop cost and context-window decay.

This combined design occupies a gap between traditions. DBMS benchmarks contribute control and repeatability but omit agent loops, while MCP and agent benchmarks contribute realistic orchestration but often omit low-level attribution and workload control (Luo et al., 2025a; Guo et al., 2026; Fan et al., 2025; Wang et al., 2026). Integrating both in one framework enables stronger claims than either tradition alone, and creates a reusable template beyond DBMSs, where protocol tax may stay small for low-volume tools but dominate for high-volume data systems.

The workload argument follows. TPC-H is the data-volume stress instrument because it preserves schema and query templates while scaling output volume, isolating middleware sensitivity to bytes and rows. SQLyzr contributes category-level complexity analysis in agent-present runs, revealing where metadata gaps trigger retries disproportionately (Abedini, 2026). Spider 2.0 and SQLStorm remain valuable references for reasoning and

broad-system evaluation, but their multi-variable shifts make them less suitable as sole controlled tracks for coupling attribution here.

Research gap statement. Existing trace-first agent benchmarks quantify protocol versus inference cost in general workflows, but no benchmark isolates MCP-to-DBMS coupling across heterogeneous DBMS backends under OLAP-scale payload stress, attributes where protocol tax transitions from secondary to dominant, and validates those findings across both agent-free controlled measurements and agent-present applied workflows (Wang et al., 2026; Transaction Processing Performance Council, 2022; Luo et al., 2025a; Guo et al., 2026; Schmidt et al., 2025).

This thesis addresses that gap through four integrated contributions: a two-track experimental design, a two-tier attribution framework, protocol-level distributed tracing with DBMS-aware span semantics, and empirical cross-coupling analysis under OLAP scale sweeps with complexity-stratified validation. If protocol-path overhead exceeds raw execution time at practical OLAP scales, the evidence will indicate that optimization priorities in agentic data systems should shift toward middleware engineering and interface design alongside traditional DBMS tuning.

The protocol-tax concept is intentionally general, since any MCP-mediated workflow incurs discovery, invocation, serialization, and parse costs. DBMS couplings make these costs highly visible because output cardinalities and schema payloads grow rapidly, so the DBMS case is a useful stress test: a benchmark that detects and attributes overhead under extreme payload pressure is better positioned to explain lighter workloads where overhead stays subdominant.

Two scope boundaries are worth stating. The attribution model is directional rather than absolute: some implementation-level interventions (schema caching, response-shape compression) indirectly influence coupling behaviour, and coupling-level constraints can motivate implementation adaptations, so the two tiers identify where primary causality appears strongest under controlled evidence while acknowledging cross-layer interaction. And if protocol tax dominates at OLAP scales, that does not imply equal dominance in low-volume transactional queries or short tool exchanges. The contribution lies in identifying where dominance emerges and why, not in asserting universal behaviour.

Three observations from historical DBMS-evaluation practice sharpen this. First, benchmark influence is strongest when metrics map to actionable design levers (Transaction Processing Performance Council, 2022); the implementation-versus-coupling split is how measured outcomes become engineering targets, which endpoint success rates and average latency alone cannot supply. Second, protocol evaluation should treat degradation patterns as primary evidence: under light workloads many couplings look similar, and divergence

emerges under stress as scale factors, payloads, and concurrent sessions reveal how quickly each loses efficiency (Transaction Processing Performance Council, 2022; Sigelman et al., 2010; Wang et al., 2026). Third, reproducibility depends on richer artifact reporting than is currently common (Wang et al., 2026; Luo et al., 2025a; Guo et al., 2026): reproducing a coupling result requires MCP server version, tool-catalog structure, schema-response policy, serialization mode, runtime language version, and trace instrumentation boundaries, and missing any can invalidate cross-study comparison even when nominal task settings match.

The broader implication reshapes how performance engineering for agentic data systems should be prioritized. Prevailing tuning starts from model selection and prompt policy (Liu et al., 2026), and AI-NativeBench's trace-first measurement supports that ordering for general workflows, where inference accounts for 86.9% to 99.9% of end-to-end latency (Wang et al., 2026). The evidence reviewed here suggests the reverse can hold under data-intensive workloads: middleware choices around schema exposure, lifecycle management, serialization format, and caching can move protocol tax onto the critical path once payloads grow into OLAP territory (Akillioglu et al., 2025; Luo et al., 2025a). This does not diminish model quality; it reframes optimization as a layered problem where interface engineering is co-equal with reasoning strategy.

The contribution is therefore is more than a benchmark; we propose a method that joins DBMS-benchmarking rigour to agent-system realism, letting future work compare couplings without confusing protocol mechanics, implementation policy, and integrated-stack effects. Negative results also become more useful, because a failed coupling can be diagnosed in terms of specific spans and metric tiers rather than dismissed as generic agent unreliability. As MCP and similar protocols become infrastructural in data-intensive AI systems, reproducible coupling benchmarks can play the role for middleware evolution that TPC benchmarks played for DBMS optimization: shared targets, revealed bottleneck classes, and stable comparative language across academia and industry. MADBench begins that transition for agentic DBMS couplings.

Chapter 3

MADBench: Workload, Dataset, and Metrics

MADBench has three main parts: a workload, a dataset, and a set of metrics. The workload defines the requests submitted to the system. The dataset defines the data on which those requests operate. The metrics define the values recorded for each run. This chapter specifies these three parts. Chapter 4 describes the systems, execution protocols, and instrumentation that implement the specification. Chapter 5 reports the results.

3.1 Dataset

The dataset defines the populated tables on which those requests run. MADBench uses the TPC-H schema and generates it at several scale factors. The same logical data is loaded into each DBMS backend. Chapter 4 defines the backends and their execution environments.

3.1.1 TPC-H Schema

TPC-H models the data warehouse of a wholesale supplier (Transaction Processing Performance Council, 2022). Customers place orders, and each order contains line items. Each line item records a part, its supplier, and its cost. The schema contains eight tables. Table 3.1 gives the SF1 cardinalities.

Table 3.1: TPC-H schema at SF1 (Transaction Processing Performance Council, 2022). Row counts grow with scale factor except for `region` and `nation`.

Table	Rows at SF1	Contents
`region`	5	Geographic regions
`nation`	25	Nations, each assigned to a region
`supplier`	10,000	Suppliers with nation and account balance
`customer`	150,000	Customers with market segment and nation
`part`	200,000	Parts with brand, type, and retail price
`partsupp`	800,000	Supplier offers and part costs
`orders`	1,500,000	Order headers with date, priority, and status
`lineitem`	6,001,215	Order lines with quantity, price, discount, and dates

3.1.2 Data Generation and Scale Factors

The `dbgen` tool generates each TPC-H instance with a fixed seed (seed = 1). The scale factor σ approximates the raw data volume in gigabytes. MADBench uses SF1 ($\sim$1 GB), SF3 ($\sim$3 GB), and SF10 ($\sim$10 GB). Track A runs at all three scales. Track B runs at SF1 and SF3.

The scale factor changes data volume while preserving the schema and query definitions. It therefore provides a controlled volume axis. Differences caused by DBMS wire formats, drivers, and runtime behaviour are treated as coupling effects rather than dataset properties. These systems-level choices are specified in Chapter 4.

3.2 Workload

MADBench uses two workloads over the same TPC-H schema. Track A contains five SQL queries from TPC-H workload and runs without an LLM. It measures middleware behaviour under controlled changes in data volume. Track B contains thirty natural-language questions that an LLM agent answers through MCP tools. It measures agent behaviour under two levels of SQL complexity. The two tracks are compared by server, backend, and scale factor in Chapter 5.

This chapter describes the workload design and gives representative examples. Appendix A contains the complete workload. It includes the five Track A SQL statements, all thirty Track B questions, the corresponding gold SQL, and the Track B system prompt. The tagged workload files in the project repository are the machine-readable versions of the same artifacts.

3.2.1 Track A: Direct SQL Workload

Track A uses the TPC-H decision-support benchmark, which defines 22 parameterized query templates over a wholesale-supplier schema (Transaction Processing Performance Council, 2022). MADBench selects five of these templates: Q1, Q3, Q6, Q9, and Q21.

We chose these five queries for two reasons. First, they cover different query structures. Q1 and Q6 use single-table scans with aggregation. Q3 contains a three-table join, while Q9 contains a six-table join. Q21 joins four tables and uses two correlated subqueries (Transaction Processing Performance Council, 2022). Second, the result cardinality of each query remains fixed as the scale factor increases. In the executed workload, Q1 returns four rows and Q6 returns one. Q3 and Q21 use `LIMIT` clauses to cap their results at 10 and 100 rows. Q9 returns approximately 175 rows because of its nation-year grouping. As the scale factor grows, DBMS work increases while the result payload remains stable. This allows Track A to measure how fixed-size protocol work changes relative to DBMS execution. It does not measure protocol behaviour for growing result sets.

We did not run the remaining 17 templates because the complete sweep would be substantially larger. Within the main Track A grid, each additional template would add 1,080 measured runs across the six couplings, three scale factors, and two runner states. The five-query subset allowed us to use 30 repetitions per cell while covering scans, aggregation, multiway joins, and correlated subqueries. The Track A conclusions are therefore limited to these five queries. Appendix A.1 provides the complete executed SQL and identifies the small dialect changes used by each backend.

Table 3.2: TPC-H queries used in Track A and their result cardinalities in the executed workload.

Query	Characterization	SF1	SF3	SF10
Q1	Aggregation, GROUP BY on lineitem	4	4	4
Q3	3-way join, `LIMIT 10`	10	10	10
Q6	Single-table scan, scalar aggregate	1	1	1
Q9	6-way join, GROUP BY nation×year	∼175	∼175	∼175
Q21	4-way join, correlated subquery, `LIMIT 100`	100	100	100

Q1 (pricing summary). Q1 scans nearly all of `lineitem`. It aggregates quantity, price, discount, and tax by return flag and line status. The result always contains four rows. Q1 therefore places most of its work in the DBMS rather than in result transfer.

Q3 (shipping priority). Q3 joins `customer`, `orders`, and `lineitem`. It finds the ten highest-revenue orders in one market segment that were placed before a date but not

shipped by that date. The `LIMIT` bounds the result to ten rows.

Q6 (revenue forecast). Q6 scans `lineitem` and returns one value. The value is the revenue represented by a range of discounts over one year. Its one-row result keeps result-transfer work small, which makes Q6 a useful protocol-overhead baseline.

Q9 (product-type profit). Q9 joins six tables and computes yearly profit by nation for parts whose names contain a given substring. It is the widest join in the selected workload. The nation-year grouping bounds the result at approximately 175 rows.

Q21 (waiting suppliers). Q21 finds suppliers in one nation who were the only late supplier on a multi-supplier order. It uses correlated `EXISTS` and `NOT EXISTS` subqueries over `lineitem`. This gives Q21 the most complex predicate structure in Track A. Its `LIMIT` caps the result at 100 rows.

All five queries keep result cardinality fixed as the scale factor grows. This design separates growth in DBMS execution time from growth caused by larger result payloads. It also creates a limitation: the workload cannot directly locate the growing-cardinality crossover (see Lemma 3.1 in Section 3.3.1).

3.2.2 Track B: agentic question workload

Track B uses the SQLyzr taxonomy to group questions by SQL complexity (Abedini, 2026). SQLyzr analyzes the abstract syntax tree of a query and assigns one of six categories, C1 through C6. The categories introduce more complex SQL constructs as the category number increases. Table 3.3 summarizes the taxonomy and shows the categories used in this thesis.

Table 3.3: SQLyzr categories used to stratify the Track B workload (Abedini, 2026).

Cat.	SQL constructs	Questions	Expected retry sensitivity
C1	Basic SELECT, single table	not used	Low
C2	WHERE, ORDER BY, GROUP BY, LIMIT, DISTINCT, equality JOIN	15	Low to moderate
C3	Non-equality JOIN, BETWEEN, LIKE, IS NULL, HAVING	not used	Moderate
C4	Single-level subqueries, UNION, INNER JOIN, 3+ table JOIN	not used	Moderate to high
C5	Subquery + GROUP BY/JOIN, two-level subqueries, CASE	15	High
C6	CTEs, window functions, 3+-level subqueries	not used	High

The executed workload uses two selected categories, C2 and C5. It contains fifteen questions from each category. C2 represents common filtering, grouping, ordering, and simple joins. C5 adds nested queries, combinations of subqueries with grouping or joins, and CASE expressions. This comparison tests whether structural complexity changes the amount of tool use and context consumed by the agent.

This thesis uses SQLyzr only for complexity stratification. It does not use SQLyzr's correctness or efficiency metrics (EA, EM, REA, CC, and ETC). MADBench uses the metrics defined in Section 3.3.2.

Question Construction

Each Track B workload item contains a natural-language question, a SQLyzr category, and a gold SQL query. The questions are written for the TPC-H schema and fixed before any model call. The gold query is written and checked at the same time as the question.

The questions are written by hand for two reasons. First, existing text-to-SQL corpora do not use the TPC-H schema at controlled scale factors. Second, hand construction assigns each question to one target SQLyzr category. This keeps the category comparison focused.

A C2 item may require one aggregate over one table:

What is the total value of all orders in the database?

```
SELECT SUM(o_totalprice) FROM orders
```

A C5 item adds a subquery before grouping:

For each part manufacturer, count parts whose retail price is above the overall average retail price.

```
SELECT p_mfgr, COUNT(*) FROM part
WHERE p_retailprice > (SELECT AVG(p_retailprice) FROM part)
GROUP BY p_mfgr
```

The first item uses a single aggregate. The second evaluates a subquery and then groups its result. This is the structural difference tested by Track B. Appendix A.2 lists all fifteen C2 questions and their SQLyzr subcategories. Appendix A.3 provides the corresponding C5 workload. Appendix A.4 gives the gold SQL for all thirty questions. The complete system prompt appears in Appendix A.5. Chapter 4 describes how the framework executes and verifies these artifacts.

3.3 Evaluation Metrics

The metrics have two purposes. First, they divide an MCP-mediated database interaction into measured timing phases. Second, they assign derived measurements to either the MCP server implementation or the integrated DBMS coupling. Chapter 4 describes how the framework records the required spans and attributes.

3.3.1 Protocol Tax Decomposition

Timing Model

Let T_{total} be the wall-clock time of one MCP-mediated database interaction. The interaction is divided as follows:

$$T_{\text{total}} = T_{\text{DB}} + \underbrace{T_{\text{disc}} + T_{\text{life}} + T_{\text{hand}} + T_{\text{ser}} + T_{\text{parse}} + T_{\text{xport}} + T_{\text{teardown}} + \epsilon}_{\text{Protocol Tax } (\Delta P)}. \tag{3.1}$$

T_{DB} is the database execution time reported by the DBMS. It includes query planning and engine execution. The seven protocol phases are discovery (T_{disc}), lifecycle management (T_{life}), handover (T_{hand}), serialization (T_{ser}), parsing (T_{parse}), transport (T_{xport}), and teardown (T_{teardown}). The residual ϵ captures timing error and unmeasured activity, such as context switching, garbage collection, or instrumentation overhead.

Definition 3.1 (Protocol Tax)**.** *For an interaction with total time T_{total} and database execution time T_{DB},* ***Protocol Tax*** *is*

$$\Delta P = T_{total} - T_{DB} = T_{disc} + T_{life} + T_{hand} + T_{ser} + T_{parse} + T_{xport} + T_{teardown} + \epsilon. \tag{3.2}$$

Definition 3.2 (Protocol Tax Ratio)**.** *The* ***Protocol Tax Ratio*** *is*

$$\Pi = \frac{\Delta P}{T_{DB}}. \tag{3.3}$$

When $\Pi < 1$, database execution takes more time than the protocol path. When $\Pi > 1$, the protocol path takes more time. RQ1 studies the crossover at $\Pi = 1$.

Definition 3.3 (Amortized Protocol Tax)**.** *Consider a session of n queries that share one discovery phase and one lifecycle initialization. The* ***amortized Protocol Tax*** *per query is*

$$\Delta P_{amort}(n) = \underbrace{\frac{T_{disc} + T_{life}}{n}}_{\text{amortized startup cost}} + \underbrace{T_{hand} + T_{ser} + T_{parse} + T_{xport} + T_{teardown}}_{\text{per-query protocol cost}}. \tag{3.4}$$

As n increases, the startup terms contribute less to each query. A long-lived client can therefore amortize discovery and connection setup. A single-question interaction cannot.

Phase definitions

Table 3.4 defines the boundaries of T_{DB} and the seven protocol phases. Chapter 4 maps these boundaries to OpenTelemetry spans.

Table 3.4: Timing phases used in the Protocol Tax decomposition.

Phase	Start event	End event	Unit	Secondary value
Discovery (T_{disc})	Client sends `tools/list`	Client receives tool manifest	ms	tokens, bytes
Lifecycle (T_{life})	Connection or pool acquisition begins	Connection becomes query-ready	ms	calls, cold/warm
Handover (T_{hand})	`tools/call` is dispatched	Server begins SQL dispatch	ms	request bytes
DB execution (T_{DB})	DBMS receives query	First result byte is available	ms	rows, plan time
Serialization (T_{ser})	First result byte is available	JSON encoding completes	ms	rows, output bytes
Parsing (T_{parse})	Client receives first response byte	Client materializes the result	ms	bytes, objects
Transport (T_{xport})	First byte leaves sender	Last byte reaches receiver	ms	bytes, mode
Teardown (T_{teardown})	Final parse span ends	Interaction span closes	ms	cleanup activity

Five rules define the accounting boundary:

1. Agent reasoning is excluded. Track A has no agent. Track B records model reasoning separately from ΔP.

2. Query planning is part of T_{DB}.

3. Serialization begins when the database driver makes the first result byte available to the MCP server.

4. Transport records IPC time for `stdio` and network-transfer time for HTTP-based transports.

5. Teardown includes cleanup between the end of parsing and the close of the interaction. It is a protocol phase rather than part of the residual.

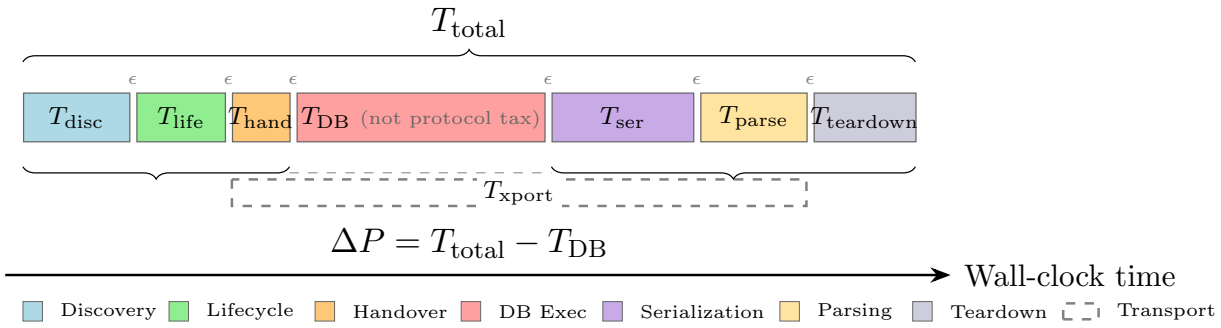


Figure 3.1: End-to-end timing model for a single MCP-mediated database interaction. The nine timing terms are the eight measured phases (T_{disc}, T_{life}, T_{hand}, T_{DB}, T_{ser}, T_{parse}, T_{xport}, T_{teardown}) plus the residual ϵ. Protocol Tax ΔP is the sum of all measured phases except T_{DB}. Transport (T_{xport}) is shown as a dashed bar because it may temporally overlap with handover and parse phases.

Decomposition properties

The timing model uses three properties to keep the phase measurements valid and comparable. Completeness detects unmeasured time. Exclusivity prevents double-counting. Observability ensures that each boundary comes from a recorded event. Chapter 4 explains how the framework checks these properties.

Property 3.1 (Completeness)**.** *The measured phases must account for nearly all of the interaction time. A trace satisfies this property when*

$$|\epsilon| < 1\,\text{ms} \quad \textit{or} \quad \frac{|\epsilon|}{T_{total}} < 0.05. \tag{3.5}$$

Completeness is needed to detect activity that is not assigned to a phase. The relative threshold limits the unaccounted share of longer traces. The absolute threshold avoids rejecting short traces because of submillisecond timing noise. A large residual means that the phase totals do not fully explain the interaction and may hide part of the Protocol Tax. The benchmark flags traces that fail this check rather than removing them silently.

Property 3.2 (Exclusivity)**.** *The measured phase intervals must be pairwise disjoint:*

$$[s_i, e_i) \cap [s_j, e_j) = \emptyset \quad \forall\, i \neq j.$$

Exclusivity is needed to prevent the same elapsed time from being assigned to more than one phase. If two measured intervals overlap, their sum can exceed the actual wall-clock time and overstate Protocol Tax. Chapter 4 explains how the framework assigns time when transport, handover, and parsing activity overlap.

Property 3.3 (Observability)**.** *Every phase boundary must correspond to a recorded start or end event. Each event is recorded with a monotonic clock. Observability is needed because phase durations must come from measured events rather than estimates. It also allows the same boundary rules to be applied across implementations. Without observable boundaries, phase comparisons cannot be reproduced reliably.*

Together, these properties make the decomposition consistent across implementations. For any set of phases $S \subseteq \{1, \ldots, k\}$, the sum $\sum_{i \in S} T_i$ equals the exclusive time measured by the corresponding spans. For the full interaction, the residual ϵ records the difference between T_{total} and the accounted phase time.

Serialization Dominance at Scale

Serialization processes every returned row. Its cost can therefore grow linearly with result cardinality. Database execution may grow more slowly when indexing, parallelism, or caching reduces the work added at each scale. The following lemma states the condition under which serialization must eventually exceed database execution.

Lemma 3.1 (Serialization Dominance at Scale)**.** *Consider a query whose result cardinality N_{rows} grows with scale factor σ. Assume that*

1. $T_{ser} = c_1 N_{rows}$ *for some* $c_1 > 0$*;*
2. $T_{DB} \leq c_2 N_{rows}^{1-\alpha}$ *for some* $c_2 > 0$ *and* $\alpha > 0$*; and*
3. $\Delta P \geq T_{ser}$ *for the measured interaction.*

Then a finite row count N_{rows}^* *exists beyond which* $\Pi > 1$*.*

Proof. The third assumption gives

$$\Pi = \frac{\Delta P}{T_{\text{DB}}} \geq \frac{T_{\text{ser}}}{T_{\text{DB}}}.$$

Applying the first two assumptions gives

$$\Pi \geq \frac{c_1 N_{\text{rows}}}{c_2 N_{\text{rows}}^{1-\alpha}} = \frac{c_1}{c_2} N_{\text{rows}}^{\alpha}.$$

Because $\alpha > 0$, this lower bound grows with N_{rows}. It exceeds 1 when

$$N_{\text{rows}} > N_{\text{rows}}^* = \left(\frac{c_2}{c_1}\right)^{1/\alpha}.$$

Therefore, $\Pi > 1$ for all row counts above this sufficient threshold. The corresponding scale factor depends on the query and the TPC-H data-generation function. □

The lemma applies only when result cardinality grows with scale. The five Track A queries in Table 3.2 do not meet this condition. They instead hold serialization volume approximately fixed while database work grows. A growing-cardinality query is required to measure the crossover directly. The result also depends on $\alpha > 0$. A full scan with nearly linear growth may weaken or remove the predicted dominance.

Derived quantities

Five quantities are derived from the phase timings and payload measurements. They are used throughout Chapter 5.

Definition 3.4 (Payload Expansion Ratio)**.**

$$\rho = \frac{B_{MCP}}{B_{native}}. \tag{3.6}$$

B_{MCP} is the byte count of the MCP JSON-RPC response. B_{native} is the byte count of the same logical result in the native DBMS format. This is a Coupling Metric because the denominator depends on the backend protocol.

Definition 3.5 (Token Inflation Factor)**.**

$$\phi = \frac{\tau_{disc} + \tau_{result} + \tau_{boiler}}{\tau^*_{result}}. \tag{3.7}$$

τ_{disc} is the discovery-response token count. τ_{result} is the result-payload token count, and τ_{boiler} is the JSON-RPC envelope cost. The denominator τ^*_{result} contains only the column values separated by whitespace. All counts use the `cl100k_base` tokenizer. This is an Implementation Metric because the server controls its schema descriptions and output format.

Definition 3.6 (Serialization Throughput)**.**

$$S_{ser} = \frac{N_{rows}}{T_{ser}} \quad \textit{(rows per millisecond).} \tag{3.8}$$

Serialization Throughput is the number of rows converted from the native result to JSON per millisecond. It is a Coupling Metric because it depends on the backend format, driver, runtime, and JSON encoder.

Definition 3.7 (Cold-to-Warm Delta)**.**

$$\delta_{CW} = T^{cold}_{life} - T^{warm}_{life}. \tag{3.9}$$

A large δ_{CW} indicates high startup cost. This is an Implementation Metric because connection reuse, pooling, and warmup behaviour are controlled by the server design.

Definition 3.8 (Context-Window Decay)**.**

$$\gamma = \frac{\tau_{disc} + \tau_{boiler}}{W_{ctx}}. \tag{3.10}$$

W_{ctx} is the model context-window size in tokens. The metric gives the fraction used by discovery text and protocol boilerplate before result content is considered. It is an Implementation Metric.

For a multi-turn Track B interaction, the protocol text accumulates. Let γ_t be the non-semantic token fraction added during tool-call round t. The cumulative value after T rounds is

$$\Gamma_T = \sum_{t=1}^{T} \gamma_t. \tag{3.11}$$

Γ_T estimates the context budget consumed by protocol text across the interaction. Chapter 5 reports Γ_T for Track B.

3.3.2 Implementation and Coupling Metrics

The benchmark assigns each metric to the source that explains most of its variation under controlled conditions. The assignment identifies the main engineering lever. It does not imply that the other layer has no effect.

Attribution criteria

Definition 3.9 (Implementation Metric)**.** *A metric m is an **Implementation Metric** if changing the MCP server code can change its value while the DBMS, runtime, hardware, and workload remain fixed. Formally, m is implementation-separable if there exist server implementations s_1 and s_2 on the same backend d, query q, and scale σ such that*

$$m(s_1, d, q, \sigma) \neq m(s_2, d, q, \sigma).$$

Definition 3.10 (Coupling Metric)**.** *A metric m is a **Coupling Metric** if its value depends on the combined behaviour of the DBMS, wire protocol, driver, runtime, and MCP server. No change to one component explains the full observed variation.*

Implementation metrics

Table 3.5 lists ten measurements tied mainly to MCP server decisions.

Table 3.5: Implementation Metrics and their main developer levers.

ID	Metric	Symbol	Unit	Track	Developer lever
IM-1	Token Inflation Factor	ϕ	dimensionless	A+B	Output format, envelope design
IM-2	Payload Size	B_{MCP}	bytes	A	JSON shape, key verbosity
IM-3	Parse Latency	T_{parse}	ms	A	Response structure
IM-4	Discovery Latency	T_{disc}	ms	A	Tool count, schema verbosity
IM-5	Lifecycle Overhead	T_{life}	ms	A	Connection model, pooling
IM-6	Cold-to-Warm Delta	δ_{CW}	ms	A	Warmup, caching
IM-7	Retry-Loop Proxy	R	count	B	Tool design, metadata completeness
IM-8	Context-Window Decay	γ	dimensionless	B	Discovery text, boilerplate
IM-9	Discovery Token Count	τ_{disc}	tokens	A	Tool descriptions, schema depth
IM-10	Lifecycle Call Count	n_{life}	count	A	Implicit or explicit lifecycle

The implemented IM-7 value is

$$R = n_{\text{tool_calls}} - 1, \tag{3.12}$$

where the subtraction removes the terminal `submit_answer` call. Thus, R counts all MCP tool calls before submission. It includes planned schema discovery and query execution as well as corrective calls. It is therefore a proxy for interaction overhead, not a direct count of failed retries. Richer foreign-key and column metadata may reduce grounding errors and unnecessary tool calls (Shi et al., 2026).

Coupling Metrics

Table 3.6 lists six measurements that depend on the complete DBMS-driver-runtime-server stack.

CM-2 varies across backends because their interfaces and result representations differ. PostgreSQL can return fields in text or binary format. MySQL's text protocol represents row values as strings, while DuckDB uses an in-process columnar representation (PostgreSQL Global Development Group, 2025; Oracle Corporation, 2025a; DuckDB Foundation, 2025). The same logical result can therefore have a different expansion ratio in each coupling.

CM-3 is the maximum resident set size during a query, sampled every 10 ms. CM-4 is the first tested scale factor at which serialization uses more than 90% of one CPU core.

Table 3.6: Coupling Metrics and their main sources of variation.

ID	Metric	Symbol	Unit	Coupling source
CM-1	Serialization Throughput	S_{ser}	rows/ms	Wire format, JSON encoder, runtime
CM-2	Payload Expansion Ratio	ρ	dimensionless	JSON shape, native encoding
CM-3	Peak Memory	M_{peak}	MB	Buffering, runtime GC, result size
CM-4	CPU Saturation Threshold	σ^*_{CPU}	scale factor	Serialization load, runtime scheduling
CM-5	Connection Handover	T_{pool}	ms	Driver pool, DBMS authentication
CM-6	Back-Pressure Pattern	β	classification	Server buffering, DBMS flow control

CM-6 classifies large-result behaviour as full buffering, chunked streaming, truncation, or an overflow error. MotherDuck's 1,024-row cap is an example of truncation (MotherDuck, 2025).

Cross-tier Interactions

Three metrics have an important secondary dependence. Serialization throughput (CM-1) depends partly on the server's JSON encoder. Payload expansion (CM-2) depends partly on the selected output format. The rank interpretation of IM-7 depends on tool-call latency within each comparison cell. Result tables mark these cases explicitly.

Safety mechanisms are measured inside the phase where they execute. Validation contributes to T_{hand}, credential setup contributes to T_{life}, and a restricted tool surface may increase R. The benchmark does not define a separate safety-overhead metric because this work cannot be separated from the corresponding protocol phase.

The workload, dataset, and metrics now define what MADBench measures. Chapter 4 defines how the framework executes this specification and records each value.

Chapter 4

Benchmarking Framework

Chapter 3 defines the MADBench workload, dataset, and metrics. This chapter explains how the framework executes that specification. It describes the systems under test, the trace model, the two execution protocols, the correctness verifier, and the analysis procedure. Chapter 5 reports the results.

The chapter describes the study that was executed. It does not include servers, query categories, or measurement stages that were only considered for later versions of the benchmark. This keeps the method consistent with the experimental grid reported in Chapter 5.

4.1 Framework overview

MADBench follows the same pipeline for both tracks. First, the framework loads one workload item and selects a compatible MCP server and DBMS backend. It then starts the required processes and discovers the server's tools. Track A submits fixed SQL, while Track B lets an LLM agent choose and invoke tools. The framework records the interaction as a trace, computes the metrics from Chapter 3, and writes the raw and derived data to disk.

The framework has three design goals. First, the same phase boundaries must apply to every server. Otherwise, phase timings cannot be compared. Second, the agent must be removable from the execution path. This allows Track A to measure middleware without model variance. Third, each run must produce enough information to reproduce the configuration and inspect the result.

These goals lead to a modular design. Server adapters provide a common interface for discovery, connection setup, and tool calls. Backend adapters load the TPC-H data and execute native SQL. A shared runner controls the experimental grid. Track A writes OpenTelemetry spans for each protocol phase. Track B writes one event log for each agent attempt. Both formats use the same server, backend, workload, scale-factor, and repetition identifiers. The analysis layer uses these fields to join the two tracks.

4.2 Systems under test

The experimental grid contains three MCP servers and three DBMS backends. Not every server supports every backend. The supported combinations form six measured cells.

4.2.1 MCP server selection

The selected servers differ in tool surface, runtime, metadata exposure, and connection policy. These differences are useful because they represent implementation choices identified in Section 2.3. Table 4.1 summarizes the three servers as executed.

Table 4.1: MCP servers used in the study. Tool counts are the counts exposed to the client during execution.

Server	Tools	Runtime	Connection and metadata policy
`anthropic-pg`	2	Node.js	PostgreSQL only; minimal query surface and basic schema resources; connection pool reuse (Anthropic, 2024)
`dbhub`	2	Node.js	Gateway for PostgreSQL and MySQL; deferred connection; progressive object search (Bytebase, 2025)
`mcp-alchemy`	4	Python	SQLAlchemy adapter for PostgreSQL, MySQL, and DuckDB; dedicated schema tools with foreign-key metadata (Kaagaard, 2025)

The comparison does not treat tool count as a complete description of a server. Two servers may expose the same number of tools but differ in startup cost, schema content, or response format. MADBench therefore records tool count together with discovery size, lifecycle time, payload size, and agent tool use.

4.2.2 DBMS backends and measured grid

The backends are PostgreSQL 16, MySQL 8.4, and DuckDB 1.0. PostgreSQL and MySQL run as separate server processes. DuckDB runs in process through its Python and SQLAlchemy interfaces. This distinction is part of the coupling under study because it changes connection setup, data conversion, and transport behaviour (PostgreSQL Global Development Group, 2025; Oracle Corporation, 2025a; DuckDB Foundation, 2025).

Table 4.2 shows the six supported server-backend cells. Both tracks use the same grid. This allows the analysis to compare controlled protocol measurements with agent outcomes without changing the server or backend.

Table 4.2: Server-backend combinations used by MADBench.

	PostgreSQL	**MySQL**	**DuckDB**
`anthropic-pg`	✓		
`dbhub`	✓	✓	
`mcp-alchemy`	✓	✓	✓

All servers use `stdio` transport from the same Python client. PostgreSQL and MySQL run in Docker containers. DuckDB uses a local database file. The framework uses the same TPC-H data, query text, and scale factor within each comparable cell.

4.2.3 Hardware and software environment

The reported measurements run on one Linux host with 32 Intel Xeon-class x86-64 cores, 128 GB of RAM, and NVMe storage. The host runs Ubuntu 22.04 with kernel 5.15, CPython 3.11.11, and Docker Engine 24. PostgreSQL 16.3 listens on port `55432`, and MySQL 8.4 listens on port `53306`. Both run in containers. DuckDB 1.0 runs in the client process. The host has no GPU. Track B model inference is the only remote component.

The client, MCP servers, and DBMS processes are otherwise colocated. As a result, T_{xport} measures local process communication rather than wide-area network delay. Each run executes one query at a time. The study therefore measures per-query cost, not concurrent-load behaviour.

4.3 Instrumentation schema

Track A uses OpenTelemetry to record the phases defined in Section 3.3.1 (OpenTelemetry Authors, 2025). The design follows the request-level span model introduced by Dapper (Sigelman et al., 2010). In a request-level span model, one trace groups all the spans created while handling one request.

In MADBench, each measured repetition produces one trace. Its root span measures the full interaction, from server setup to cleanup. Child spans measure the individual steps within that interaction. These steps include discovery, lifecycle setup, handover, serialization, parsing, and transport. Each span records when the step starts and ends. It also records which parent span it belongs to. MADBench uses these records to determine how long each step takes and where steps overlap.

The database reference time is stored as a trace attribute. Teardown is calculated from the end of the final parsing span to the end of the root span. This information allows MADBench to calculate the phase durations without counting the same time twice.

Track B uses a simpler event trace. Its main outputs are correctness, tool use, tokens, cost, and total attempt time. Therefore, it does not provide the same phase-level span tree as Track A.

4.3.1 Span hierarchy and timing boundaries

The Track A root span is `mcp.interaction`. It begins before server initialization and ends after the client has parsed the response and completed cleanup. The span stores the server, backend, query, scale factor, and track. The cell directory stores the cold or warm state. The trace loader assigns the repetition index from the order of root spans in the file.

Discovery records the `tools/list` exchange. Lifecycle records subprocess startup, MCP initialization, and database connection setup. Handover records request construction before the tool call. Serialization records the outer conversion stage visible to the adapter. Parsing records client-side materialization of the response. Transport records the `stdio` round trip. Teardown is derived from the end of the final parse span to the end of the interaction.

The serialization boundary is not complete for every upstream server. In particular, per-row string conversion can occur before the adapter's serialization span opens. That work then appears inside transport. Therefore, T_{ser} should be read as the measured outer encoding cost, not as the full cost of converting every database value to text. Chapter 5 keeps this limitation explicit when interpreting the phase shares.

Transport may overlap with handover or parsing when a server buffers or streams data. The analysis removes these overlaps before adding transport to Protocol Tax:

$$T^{\text{excl}}_{\text{xport}} = T^{\text{wall}}_{\text{xport}} - T_{\text{xport}\cap\text{hand}} - T_{\text{xport}\cap\text{parse}}. \tag{4.1}$$

This correction enforces the exclusivity requirement in Property 3.2. It prevents the same wall-clock interval from being assigned to two phases.

4.3.2 Database-time reference

The upstream MCP servers do not expose DBMS execution time in a common response field. MADBench therefore measures T_{DB} through a harness-managed native connection. The harness executes the same SQL against the same database instance immediately before the MCP tool call. Query planning is included in this measurement.

This design provides one consistent database-time reference across servers. However, it is not the exact execution time observed inside the MCP server. The native call and the server call use separate connections, and the first call may affect the shared cache. The transport span also includes the server's own query execution while the client waits for the response. MADBench records the native T_{DB} value on the transport trace so this relationship remains visible during analysis. Chapter 5 treats the reference path and execution order as measurement limits.

The native call used for T_{DB} does not provide B_{native}. Therefore, it cannot be used to compute the Payload Expansion Ratio in Definition 3.4. That metric requires a separate native-byte measurement.

4.3.3 Span attributes

Each span carries the attributes needed to normalize and filter the measurements. Table 4.3 gives the main fields. Raw SQL and result data are retained only where required for validation. The aggregate tables use identifiers, timings, counts, and payload sizes.

The fields are not all available for every server. For example, resource sampling is incomplete for some cells, and discovery tokens require a tokenizer pass. Chapter 5 distinguishes measured values from proxies and unavailable metrics.

Table 4.3: Main attributes recorded by the trace pipeline.

Span or record	Main attributes
`mcp.interaction`	server, backend, Track A query identifier, scale factor, track
`mcp.discovery`	response bytes, response tokens when available, tool count
`mcp.lifecycle`	cold or warm state, lifecycle call count
`mcp.tool_call`	tool name, request bytes, response bytes, row count
`mcp.serialization`	row count, measured database reference time, output size when available
`mcp.parse`	input bytes, row count, resource samples when available
`mcp.transport`	payload bytes, transport mode, database reference time
Track B attempt record	question identifier, SQLyzr category, scale factor, repetition, model turns, tool calls, tokens, final answer, terminal outcome, wall-clock time

4.3.4 Exclusion and calibration rules

Three rules keep the accounting boundary consistent. First, model inference is not part of Protocol Tax. Track B wall-clock time includes remote model calls, but it is not used as a Protocol Tax measurement. Second, query planning is included in T_{DB}. Third, cleanup after parsing is assigned to T_{teardown}.

The trace is then checked against the three properties from Section 3.3.1. Completeness compares the root duration with the sum of the measured phases. Exclusivity checks for overlapping phase intervals. Observability checks that each boundary has a recorded system event and a monotonic timestamp. A failed property check is retained as a diagnostic unless an explicit exclusion rule applies. It is not silently removed from the corpus.

4.3.5 Calibration procedure

The framework defines an observer-effect check by comparing traced and untraced executions of the same cell. Let T_{total} be the traced or untraced wall-clock time. The observer overhead is

$$\delta_{\text{obs}} = \frac{T_{\text{total}}^{\text{traced}} - T_{\text{total}}^{\text{untraced}}}{T_{\text{total}}^{\text{untraced}}}. \tag{4.2}$$

The target threshold is 5%. However, the reported campaign does not contain a separate

untraced calibration sweep. It evaluates trace closure through the completeness, exclusivity, and observability checks instead. Short DuckDB calls may be close to the timer and exporter noise floor. Chapter 5 therefore also reports an absolute residual threshold of 1 ms. This does not replace a direct observer-effect measurement, and the sub-millisecond results should be read with that limit in mind.

All timestamps use a monotonic clock. The framework records the repetition index and the seed `20260525`. The executed cells and repetitions run sequentially. The seed is stored in the manifest, but the reported campaign does not shuffle the execution order.

4.4 Experimental protocols

Both tracks use the same server-backend grid and common identifiers, but they use different raw trace formats. Track A uses fixed SQL from the benchmark. Track B begins with a natural-language question and lets an agent decide which tools to call.

Table 4.4 summarizes the executed campaign. It separates the number of experimental cells from the number of traces. A Track A cell fixes the server, backend, query, scale factor, and cold or warm state. A Track B attempt additionally fixes one natural-language question and one repetition.

Table 4.4: Executed experimental configuration.

Setting	**Track A**	**Track B**
Input	TPC-H Q1, Q3, Q6, Q9, Q21	30 fixed questions: 15 C2 and 15 C5
Scale factors	SF1, SF3, SF10	SF1, SF3
Repetitions	30 per cold or warm cell	3 per question and scale factor
Warmup	5 discarded runs per cell	none
Cache or process state	separate cold and warm runner states	no cold/warm factor; PostgreSQL restarted before each attempt
Original-schema size	180 cells; 5,400 planned traces	1,080 planned attempts
Additional run	90-cell `mcp-alchemy` index sensitivity; 2,700 traces	none

4.4.1 Track A: controlled measurement

Track A runs TPC-H Q1, Q3, Q6, Q9, and Q21 at SF1, SF3, and SF10. Appendix A.1 contains the executed SQL. For each supported server-backend pair, every query is run under separate cold and warm conditions. Cold and warm each have 30 measured repetitions after five discarded warmup runs. Thus, one server-backend-query-scale tuple has 60 measured runs: 30 cold and 30 warm. The complete original-schema grid has $6 \times 5 \times 3 \times 2 = 180$ cells and 5,400 planned runs.

The grid is split by server support. `mcp-alchemy` contributes 90 cells across three backends, or 2,700 planned traces. `dbhub` contributes 60 cells across two backends, or 1,800 planned traces. `anthropic-pg` contributes 30 PostgreSQL cells, or 900 planned traces.

The runner also produces 900 warmup interactions, five for each of the 180 cells. These are stored separately and are not analyzed. Chapter 5 reports the final trace count and explains any incomplete or excluded runs.

One Track A repetition follows five steps:

1. Start the required MCP server and backend state.
2. Call `tools/list` and record discovery.
3. Execute the native reference query and record T_{DB}.
4. Submit the fixed SQL through the MCP query tool and record handover, transport, serialization, and parsing.
5. Close the interaction, derive teardown, validate the trace, and write the result.

The SQL is fixed before execution. No model generates or modifies it. This removes model variance and makes differences attributable to the server implementation or the server-backend coupling.

A separate sensitivity run repeats the 90-cell `mcp-alchemy` grid after adding an index on `lineitem.l_shipdate` in PostgreSQL. This run uses the same five queries, three scale factors, two runner states, five warmups, and 30 measured repetitions. It contributes 2,700 measured traces and 450 discarded warmup interactions. The run tests whether reducing database execution time changes the relative Protocol Tax. Chapter 5 reports it separately from the original-schema grid.

4.4.2 Paired cold and warm runs

Track A records cold and warm runner states separately. In a cold cell, each repetition creates a fresh server-adapter object, tracer, and native backend connection. In a warm cell, one adapter object, tracer, and native backend connection are reused across the warmups and measured repetitions. The main Track A sweep does not restart a database container before every repetition. Thus, the cold/warm label primarily measures client and connection reuse. It is not a guarantee that the DBMS buffer cache is empty.

The two conditions give the Cold-to-Warm Delta from Definition 3.7. The framework does not issue cache-clearing SQL. This avoids perturbing the DBMS through an extra in-band command, but it also limits the result to the executed runner-state definition.

4.4.3 Track B: agent-present measurement

Track B uses the thirty questions defined in Section 3.2.2. Fifteen are in category C2 and fifteen are in C5. The full question set, gold SQL, and system prompt appear in Appendix A.

Track B runs only at SF1 and SF3. Each question has one fixed wording and three repetitions at each scale factor in each supported server-backend cell. The complete analysis grid therefore has $6 \times 30 \times 2 \times 3 = 1{,}080$ agent attempts. These repetitions measure consistency on one wording. They do not measure paraphrase robustness.

The server totals follow the same support matrix. `mcp-alchemy` contributes 540 attempts, `dbhub` contributes 360, and `anthropic-pg` contributes 180. Each server-backend-scale combination contains 90 attempts.

Track B does not create separate cold and warm cells. However, every PostgreSQL sweep enables the paired-cache flag. The runner restarts PostgreSQL and waits for a healthy state before each PostgreSQL attempt. MySQL and DuckDB are not restarted between attempts. This controls the starting state for PostgreSQL without adding a second warm condition.

The agent uses `deepseek/deepseek-v4-flash` through OpenRouter with temperature 0.0 and a 4,096-token output limit per model turn. The model, prompt, and decoding configuration are fixed across all cells. The runner allows ten model-and-tool loop iterations and configures a retry budget of three. One model turn can request more than one tool. Therefore, the recorded tool-call count can exceed ten even when the loop ends with `call_limit_exceeded`. A retry is the same tool being issued again after the previous call returned an error.

The executed code does not apply one 180-second deadline to the complete attempt. Instead, each OpenRouter request has a 120-second HTTP timeout. PostgreSQL and MySQL verification queries also use a 120-second statement or read timeout. These are per-operation limits, so a multi-turn attempt can last longer. The sweep runner also enforces a cost cap. The recorded caps range from USD 2 to USD 8 across the executed sweeps.

One Track B attempt follows this loop:

1. Load the system prompt and one natural-language question.
2. Discover the server tools and add a `submit_answer` tool.
3. Ask the model to select a tool and provide its arguments.
4. Execute the tool and return the result to the model.
5. Repeat until the model submits an answer or reaches a limit.
6. Compare the submitted answer with the stored gold result.

The primary IM-7 value is $R = n_{\text{tool_calls}} - 1$, as defined in Equation 3.12. It counts calls made before `submit_answer`. It is a tool-use proxy. It does not distinguish a planned schema lookup from a corrective retry.

Each attempt is written to `rep_NNN.jsonl`. The first event records the question, backend, repetition, wall-clock time, and any agent error. Later events record the final outcome, token counts, tool name, tool arguments, tool result, and retry flag. This event log is the raw Track B trace. It is separate from the OpenTelemetry span file used by Track A.

4.5 Correctness and failure classification

Track B requires a deterministic correctness check. Before the agent sweep, the framework executes each gold SQL query for the corresponding backend and scale factor. It stores the normalized columns and rows in a scale-specific gold file. The agent's submitted answer is then compared with this stored result.

4.5.1 Comparison semantics

The verifier accepts a SQL string, scalar, list of rows, or dictionary containing rows. If the answer is SQL, the verifier executes it against the same backend. Other answer shapes are converted to a common row representation. DuckDB is an exception. The `mcp-alchemy` child holds the database-file lock during an attempt, so the verifier does not open a second DuckDB SQL connection. A SQL-shaped DuckDB answer is therefore `unparseable`. Scalar and row-list answers are unaffected.

Rows are compared without order unless the gold SQL contains `ORDER BY`. Column order is matched by normalized column name when names are available. Numeric values use an absolute tolerance of 10^{-4}. Strings are compared after trimming whitespace and converting case. The primary verdict is `correct`, `incorrect`, or `unparseable`.

4.5.2 Shape drift

An answer can contain the correct values in the wrong data structure. For example, the model may submit the string `'[[42]]'` instead of the list `[[42]]`. A lenient value comparison can accept both forms, but they are not equivalent outputs for an application.

The failure analyzer therefore records `correct_with_shape_drift` as a diagnostic label. The underlying value is correct, but the submitted structure differs from the expected structure. This label is reported separately so that formatting failures are not hidden inside the correctness rate.

4.5.3 Failure-mode analyzer

The analyzer reads the raw agent trace after verification. It separates value mismatches, unparseable answers, call-limit exhaustion, agent errors, and shape drift. These labels are deterministic and do not require another model call. Failed attempts remain in the denominator for completion and failure-rate analysis. The output is stored once per attempt and is used by the Track B analysis.

4.6 Analysis procedures

The analysis first computes one record per trace. It then aggregates records within experimental cells. This order keeps the raw measurements available and prevents cell

summaries from hiding failed or excluded runs.

4.6.1 Track A analysis

Track A reports the distribution of each timing phase and the Protocol Tax Ratio. The primary cell summary is the median. P5, P25, P75, P95, and P99 describe the distribution and its tail. Phase shares show where Protocol Tax is spent within each cell.

Scale sensitivity is summarized with the fitted relationship

$$\Pi(\sigma) = \alpha\sigma^{\beta_{\text{sf}}}, \tag{4.3}$$

where σ is the TPC-H scale factor. The fit describes the direction and rate of change over SF1, SF3, and SF10. Any crossover outside this measured range is an extrapolation, not an observed result.

Implementation Metrics are compared across servers while holding the backend fixed. Coupling Metrics are compared across supported server-backend cells. This follows the attribution rules in Section 3.3.2.

4.6.2 Track B analysis

Track B reports completion, failure class, tool-call count, token use, cost, and wall-clock time. Results are grouped by server, backend, scale factor, and SQLyzr category. C2 and C5 are compared directly because these are the two executed categories.

The three repetitions measure consistency for one fixed wording. They do not measure robustness to paraphrasing. Completion rates are therefore conditional on the selected model, prompt, and question wording.

4.6.3 Cross-track analysis

The cross-track analysis joins Track A and Track B by server and scale factor. Its main comparison asks whether the server ranking by median Π agrees with the ranking by agent completion. It also compares Π with the IM-7 tool-call proxy.

Spearman's rank correlation measures whether two variables place the same cases in a similar order (Spearman, 1904). It is used when the order matters more than the distance

between the values. The coefficient ranges from -1 to $+1$. If it is $+1$, the two variables increase in the same order. If it is -1, one variable increases while the other decreases. A value near zero means that there is no consistent order in the observed data.

For a simple example, suppose server A has the lowest Π and the highest completion rate. Server B is in the middle on both measures. Server C has the highest Π and the lowest completion rate. As Π increases, completion decreases. The Spearman coefficient is therefore -1.

MADBench needs this test for the cross-track question. It checks whether servers with lower Protocol Tax in Track A also tend to achieve higher agent completion in Track B. A negative coefficient supports that pattern because lower Π is paired with higher completion. The two measures use different units, and the thesis asks about their ordering rather than a fixed numerical change. Spearman's test matches that question. It tells us whether Track A measurements may help guide server selection for agent workloads.

Only three servers appear in both tracks, and their observed ranks contain no ties. There are therefore only $3! = 6$ possible rank orders. In a two-sided exact test, even a perfect relationship has $p = 2/6 = 1/3$. The two extreme cases are a perfectly matching order and a perfectly reversed order. As a result, no Spearman result can reach $\alpha = 0.05$ with three servers. The result can describe the measured systems, but it cannot establish a general relationship across MCP servers.

4.6.4 Statistical controls

Confidence intervals for medians and other derived values use percentile bootstrap resampling with 2,000 resamples (Efron, 1979). The bootstrap treats the observed repetitions as the best available estimate of the underlying distribution. It draws 30 traces at random from the measured 30 with replacement, computes the median of that resample, and repeats this 2,000 times. The middle 95% of the resampled medians forms the confidence interval.

For example, if the resampled medians for one cell mostly fall between 1.20 and 1.31, the reported interval is $[1.20, 1.31]$. MADBench uses this method because latency distributions are skewed and contain outliers, so an interval built on a normal approximation around the mean would be misleading. The percentile bootstrap assumes nothing about the shape of the distribution. It describes the uncertainty of each cell median. It does not correct for systematic measurement error.

Completion rates use Wilson score intervals (Wilson, 1927). A completion rate is a proportion, such as 25 correct attempts out of 45. The Wilson interval places uncertainty

bounds around a proportion in a way that respects the 0–100% limits and remains reliable when the sample is small or the rate is extreme.

For example, a cell with 43 correct attempts out of 45 has a rate of 95.6%. A naive interval around that rate can extend past 100%, which is impossible. The Wilson interval stays inside the valid range. MADBench uses it because a Track B cell holds 45 attempts per category, and several cells sit near the boundaries. The interval shows how precise each completion rate is. It does not test whether two rates differ; that comparison uses the tests below.

Sign tests compare the direction of paired differences (Dixon and Mood, 1946). Each pair of matched cells either moves up or moves down between two conditions. The test counts the directions, ignores the sizes, and asks how likely the observed count is if both directions were equally probable.

For example, C2 completion exceeds C5 completion in every one of the six paired cells. If category had no effect, each cell would be a coin flip, so the sign test gives $p = 2^{-6} = 0.0156$. MADBench uses the sign test because cells differ widely in difficulty, and the direction of a difference is comparable across cells even when its size is not. Where two completion rates are compared in aggregate, a pooled two-proportion test supports the cell-level result. Effect sizes are reported with the corresponding tests so that the size of a difference is shown alongside its p-value.

4.7 Cost control and reproducibility

4.7.1 Cost tracking

Track B records provider-reported token use and attributes API cost to each attempt. The sweep runner polls the OpenRouter key endpoint and stops when the configured cap is reached. Each sweep records its cap in the manifest; the executed caps range from USD 2 to USD 8. Chapter 5 reports the observed cost per attempt and per correct answer.

4.7.2 Reproducibility artifacts

Each sweep writes a manifest. It records the server, backends, workload items, scale factor, repetitions, model settings, cache flag, software versions, and configuration. Track A stores measured spans in `trace.jsonl`. Its five discarded warmups are stored separately

in `trace_warmup.jsonl` and are not included in the reported sample. Track B stores one `rep_NNN.jsonl` file per agent attempt. Analysis scripts produce `per_trace_metrics.csv` and per-cell summaries from these raw files.

The complete workload is provided in Appendix A. The project repository also contains the machine-readable workload files, system prompt, gold results, run manifests, analysis scripts, and figure-generation code. These artifacts allow a reader to inspect the exact inputs without placing long configuration files in the chapter.

4.8 Summary

MADBench uses one six-cell server-backend grid for both tracks. Track A submits fixed SQL and measures the protocol path without an LLM. Its design contains 5,400 original-schema runs and a separate 2,700-run index sensitivity analysis. Track B uses the same systems at SF1 and SF3 with a fixed agent configuration. Its design contains 1,080 attempts. The two tracks use different raw trace formats but share the identifiers needed for analysis.

The framework also states its measurement limits. Database time is a same-instance native reference rather than the exact time inside the MCP server. Some byte, token, resource, and serialization fields are incomplete. The cross-track analysis contains only three servers. Chapter 5 reports which metrics are measured directly, which use proxies, and how these limits affect the conclusions.

Chapter 5

Experiments and Results

This chapter reports the MADBench experiments. Track A measures the cost of the MCP path with fixed SQL. Track B adds an LLM agent and measures correctness, tool use, token use, and cost. The two tracks use the same six server–backend combinations. Track A contains 5,378 usable traces, while Track B contains 1,080 agent attempts.

The results answer three questions. First, protocol cost depends strongly on the server process model and the DBMS coupling. Second, low protocol cost does not guarantee high agent completion. Third, workload scale changes both measurements. It reduces the relative effect of fixed middleware cost in Track A, but it can expose payload and context limits in Track B.

Appendix A contains the complete workload. Section A.1 lists the five Track A TPC-H queries. Sections A.2 and A.3 list the Track B questions, and Section A.4 gives their reference SQL. Chapter 4 describes the execution and analysis procedures.

5.1 Experimental corpus

The experiments are run on the hardware and software environment described in Section 4.2.3. The measured grid contains three MCP servers and three DBMS backends. Not every server supports every backend, so the grid has six valid combinations. Table 5.1 summarizes them.

Table 5.1: Server–backend combinations used in both tracks.

Server	Tools	Runtime	Backends	Connection policy
`anthropic-pg`	2	Node.js	PostgreSQL	Reuses a PostgreSQL pool during the session
`dbhub`	2	Node.js	PostgreSQL, MySQL	Opens the configured backend connection when needed
`mcp-alchemy`	4	Python	PostgreSQL, MySQL, DuckDB	Uses SQLAlchemy and exposes dedicated schema tools

5.1.1 Executed runs

Track A uses TPC-H Q1, Q3, Q6, Q9, and Q21 at SF1, SF3, and SF10. Each cold or warm cell contains 30 measured repetitions after five discarded warmup runs. This gives 180 cells and 5,400 planned traces.

However, one `dbhub`×MySQL×Q21×SF10 warm cell ended after eight measured repetitions. The `dbhub` subprocess stopped and did not recover. Before the failure, the same long-lived process had run Q21 thirteen times in a row: five warmups and eight measured repetitions. This was about thirty minutes of continuous load. Each execution took roughly 140 seconds, with about half of that time spent in MySQL.

In contrast, the paired cold cell completed all 30 repetitions. Each cold repetition starts a fresh server process. This pattern points to the long-lived process under sustained load rather than the query or MySQL alone. Restarting the server would have changed the warm condition, so the harness ended the cell. All eight recorded traces passed validation.

The final corpus contains 5,378 traces. We retain the incomplete cell with its eight observed repetitions. We do not estimate values for the missing 22 runs.

Track B uses 30 natural-language questions at SF1 and SF3. Fifteen questions are in SQLyzr category C2, and fifteen are in C5. We chose these categories on purpose. Track B asks one narrow question: does agent behaviour change between routine and complex SQL? It does not test the full SQLyzr taxonomy. A two-level comparison is enough to answer this question, and it makes the best use of the attempt budget: two categories give 45 attempts per cell, scale factor, and category, where six would leave 15 and confidence intervals too wide to separate the cells.

C2 is the lower condition. It covers filtering, aggregation, grouping, and equality joins. C5 is the higher condition. It adds nested queries, combinations of subqueries with joins or grouping, and `CASE` expressions. C1 contains basic projection only, so it does not exercise

the schema grounding studied here. C3 and C4 fall between C2 and C5. They would help describe how the effect develops, but they are not needed for this comparison. Finally, C6 adds CTEs, window functions, and deeper nesting. Including it would extend the study to advanced SQL generation, where failures reflect the model rather than the server under test. C2 and C5 also bracket the expected retry-sensitivity range in Table 3.3.

Therefore, Track B compares C2 with C5 only. Its conclusions apply to these two workloads. They do not establish a trend across all six categories. Each category contains 15 questions. Each question is repeated three times in every server–backend cell and at both scale factors. This gives $6 \times 30 \times 2 \times 3 = 1{,}080$ attempts. All attempts are present.

Table 5.2 summarizes the two tracks. The complete query text, question text, reference SQL, and system prompt appear in Appendix A.

Table 5.2: Executed experimental corpus.

Setting	**Track A**	**Track B**
Input	TPC-H Q1, Q3, Q6, Q9, Q21	15 C2 and 15 C5 questions
Scale factors	1, 3, 10	1, 3
Repetitions	30 per cold or warm cell	3 per question and scale factor
Measured cells	180	360 question-level cells
Usable observations	5,378	1,080
Model	None	`deepseek/deepseek-v4-flash`
Primary output	Phase timing and Π	Verified completion and interaction cost

5.1.2 Protocol-tax accounting

The results use the definition in Section 3.3.2. Protocol Tax includes discovery, lifecycle, handover, serialization, parsing, transport, teardown, and the remaining timing residual. Thus,

$$\Pi = \frac{T_{\text{total}} - T_{\text{DB}}}{T_{\text{DB}}}. \tag{5.1}$$

This detail matters for `anthropic-pg`. Its child process does not exit when the client closes standard input. The adapter waits about 5 seconds before terminating it. Earlier analysis tables used only the six inner protocol phases and omitted this shutdown interval. The results below include it because teardown is part of the protocol path. This makes the reported values consistent with the analysis in Chapter 3.

5.2 Track A: controlled protocol measurements

Track A removes the agent and sends fixed SQL through each MCP server. It measures how much time the middleware adds around a known database operation. This cost varies by server, backend, query, scale factor, and process state. The phase traces link much of that variation to the process and connection choices described in Chapter 4.

5.2.1 Overall Protocol Tax Ratio

Table 5.3 reports the median of the 30 repetition-level Π values in each cell. It then summarizes these cell medians for each server–backend combination. Across the full grid, the cell median ranges from 0.0031 to 7.32.

Table 5.3: Track A Protocol Tax Ratio by server–backend combination. The middle column is the median of cell medians. The range covers query, scale factor, and cold or warm state.

Server	Backend	Median Π	Minimum cell	Maximum cell
`mcp-alchemy`	DuckDB	0.019	0.003, Q21 SF10 warm	0.147, Q6 SF1 cold
`mcp-alchemy`	MySQL	1.081	1.008, Q9 SF10 cold	1.746, Q3 SF1 warm
`mcp-alchemy`	PostgreSQL	1.246	1.016, Q1 SF10 cold	2.712, Q3 SF1 warm
`dbhub`	MySQL	1.077	1.019, Q9 SF10 cold	2.231, Q3 SF1 warm
`dbhub`	PostgreSQL	1.314	1.052, Q1 SF10 cold	3.059, Q6 SF1 warm
`anthropic-pg`	PostgreSQL	1.874	1.081, Q1 SF10 cold	7.322, Q6 SF1 warm

DuckDB is the only coupling whose median remains well below one. It runs in the client process, so it avoids a separate DBMS wire exchange. In the other five combinations, the median is above one. Their protocol path therefore takes at least as much time as the native database reference in a typical cell.

The PostgreSQL comparison also shows a server effect. `mcp-alchemy` and `dbhub` have median ratios of 1.246 and 1.314. The difference is modest. `anthropic-pg` rises to 1.874 after its shutdown wait is included. Thus, teardown changes the ranking. Omitting it would make `anthropic-pg` appear to have the lowest PostgreSQL overhead.

These values compare complete server–backend couplings. They do not rank programming languages. Each server has a different process model, tool surface, driver, and response format. The experiment cannot separate those choices through language alone.

5.2.2 Phase attribution

The phase traces explain the difference between the servers. Table 5.4 gives one representative cell: PostgreSQL Q6 at SF1 in the warm condition. The values are medians over 30 repetitions.

Table 5.4: Median phase duration for PostgreSQL Q6 at SF1, warm state (ms). Tail covers teardown or the unattributed interval after the last child span.

Server	T_{disc}	T_{life}	T_{DB}	T_{ser}	T_{parse}	T_{xport}	**Tail**
`mcp-alchemy`	2.36	938.14	878.99	0.13	0.18	1006.79	151.90
`dbhub`	5.62	816.93	405.32	0.13	0.15	445.15	18.63
`anthropic-pg`	0.99	108.96	849.50	0.14	0.19	1184.86	5015.92

Discovery, outer serialization, and client parsing are small in this cell. Lifecycle and transport take most of the measured time for `mcp-alchemy` and `dbhub`. For `anthropic-pg`, teardown is the largest term. The process waits about 5 seconds before forced termination. This is a server-lifecycle behaviour, not database execution.

The serialization span needs a careful interpretation. It surrounds the outer adapter encoding step. Some row conversion occurs inside the upstream server while the client waits for the response. That work is recorded in transport rather than serialization. The sub-millisecond T_{ser} values therefore do not mean that the full result was converted to JSON in less than one millisecond. They only describe the visible outer boundary.

Handover is zero throughout the executed grid. The `stdio` client constructs and writes each request inside the transport operation, so the harness cannot isolate a separate handover interval. This is a degenerate measurement, not evidence that request handover has no cost.

Cold and warm states mainly change client and connection reuse. They do not clear the DBMS buffer cache before every repetition. Therefore, the cold–warm difference should not be described as a pure database-cache effect. It combines adapter construction, connection reuse, and shared database state.

5.2.3 Effect of scale factor

Protocol Tax becomes less important relative to database execution as the scale factor increases. The cell median falls in 119 of the 120 adjacent SF1-to-SF3 or SF3-to-SF10 comparisons. Table 5.5 shows the median at each scale factor for every coupling.

Table 5.5: Median Track A Π by scale factor. Each value summarizes query and cold or warm cell medians within one coupling.

Server	Backend	SF1	SF3	SF10
`mcp-alchemy`	DuckDB	0.030	0.020	0.011
`mcp-alchemy`	MySQL	1.220	1.086	1.033
`mcp-alchemy`	PostgreSQL	1.680	1.268	1.083
`dbhub`	MySQL	1.290	1.061	1.033
`dbhub`	PostgreSQL	2.248	1.328	1.126
`anthropic-pg`	PostgreSQL	4.055	1.972	1.279

The trend is consistent across all six couplings. Fixed startup and shutdown costs are large at SF1 because the database query is short. As the data volume grows, T_{DB} increases and amortizes those fixed costs. The absolute middleware time does not necessarily fall. Its ratio to database time falls.

This direction is important for the thesis hypothesis. Protocol cost often matches or exceeds database time, but its relative share does not grow with the tested scale factors. It falls. The experiment therefore supports protocol cost as a first-order systems concern, but it does not support a claim that the ratio increases as the OLAP dataset grows.

DuckDB follows the same direction but remains in a different range. Its in-process path has no separate DBMS network exchange. At SF10, database work accounts for almost all of the interaction time. This result is specific to the embedded coupling and should not be generalized to client–server DBMSs.

5.2.4 Trace validity

The final Track A corpus contains 179 complete 30-repetition cells and one eight-repetition cell. All observed traces are retained. No value is imputed for the incomplete cell.

Completeness uses the amended rule from Chapter 4: a trace passes when the absolute residual is below 1 ms or below 5% of T_{total}. Under this rule, 5,347 of 5,378 traces pass, or 99.42%. The remaining 31 traces come from five `mcp-alchemy` cells. Four are short PostgreSQL SF1 cells with a 150–270 ms subprocess tail. One is a DuckDB trace that misses the threshold by 0.08 percentage points. These traces are kept because the cause is known and their effect is visible in the reported residual.

The exclusivity and eight-term balance checks pass for the audited cells. The derived teardown term also closes the five-second `anthropic-pg` shutdown interval. This correction

is included in the Π values reported here.

5.2.5 Index sensitivity

A separate `mcp-alchemy`×PostgreSQL run adds an index on `lineitem.l_shipdate`. It repeats the same five queries, scale factors, process states, and 30 repetitions. The index mainly affects Q6 at SF1. Table 5.6 compares the original and indexed runs.

Table 5.6: Effect of the `l_shipdate` index on PostgreSQL Q6. Π includes the complete protocol path.

Cell	Original T_{DB} (ms)	Indexed T_{DB} (ms)	Original Π	Indexed Π
SF1 cold	1293.82	563.83	1.94	3.03
SF1 warm	878.99	362.67	2.40	4.70
SF3 cold	2959.75	3379.27	1.44	1.33
SF3 warm	2318.19	2181.89	1.93	1.70
SF10 cold	7975.81	8197.17	1.18	1.15
SF10 warm	7557.32	7334.03	1.39	1.26

At SF1, the index reduces median database time by 56% in the cold state and 59% in the warm state. The middleware path does not fall by the same amount. As a result, Π rises from 1.94 to 3.03 and from 2.40 to 4.70. The SF3 and SF10 changes are small and mixed. This result does not show that the index makes the system slower. It shows that faster SQL can make a fixed protocol cost more visible.

5.3 Track B: agent-present results

Track B tests the same six server–backend combinations with an LLM agent. The agent receives a natural-language question, discovers the available tools, executes its chosen calls, and submits an answer. The verifier compares that answer with the gold result. Completion therefore means verifier-confirmed correctness, not agent self-report.

The two Track B sweeps differ only in scale factor. Each cell contains the same 30 questions and three repetitions per question. The results first report SF1, then compare SF1 with SF3.

5.3.1 Completion at SF1

Table 5.7 reports completion, cost per correct answer, and mean tool calls for the six SF1 cells. Completion ranges from 53.3% to 65.6%. The Wilson confidence intervals overlap, so the observed sample does not establish a clear winner on completion alone. The Wilson intervals treat the 90 attempts in each cell as independent Bernoulli observations. However, each question appears three times using the same wording. The intervals are therefore descriptive for this workload. They do not estimate uncertainty over a broader population of questions.

Table 5.7: Track B results at SF1. Each cell contains 90 attempts.

Server	Backend	Completion	95% Wilson CI	USD/correct	Mean calls
`mcp-alchemy`	PostgreSQL	0.533	[0.431, 0.633]	0.00361	5.07
`anthropic-pg`	PostgreSQL	0.578	[0.475, 0.675]	0.00402	1.24
`mcp-alchemy`	DuckDB	0.600	[0.497, 0.695]	0.00327	5.23
`dbhub`	PostgreSQL	0.611	[0.508, 0.705]	0.00622	3.96
`mcp-alchemy`	MySQL	0.644	[0.542, 0.736]	0.00309	4.80
`dbhub`	MySQL	0.656	[0.553, 0.746]	0.00293	2.47

The additional metrics separate cells that have similar completion. For example, `dbhub`×PostgreSQL completes 61.1% of the attempts, but it has the highest SF1 cost per correct answer. `mcp-alchemy`×DuckDB completes 60.0%, almost the same rate, at about half the cost per correct answer. The two systems reach similar outcomes through different interaction paths.

Backend results also depend on the server. MySQL is above PostgreSQL for both `dbhub` and `mcp-alchemy` at SF1. However, the experiment does not isolate the cause. Driver behaviour, response format, database output, and model interaction all change together. The result should therefore be read as a coupling difference.

5.3.2 Failure modes

Completion alone hides how the unsuccessful attempts end. Table 5.8 divides the 540 SF1 attempts into five terminal outcomes. The classes are mutually exclusive.

The `mcp-alchemy` cells mainly fail by reaching the loop limit. The two Node.js server families mainly fail with agent errors or without a parseable submission. This is a useful diagnostic pattern, but it is not a clean language comparison. Runtime, tool count, response format, and server implementation are confounded in the observed systems.

Table 5.8: Track B outcome counts at SF1. CLE denotes call-limit exceeded.

Server	Backend	Correct	Incorrect	CLE	Agent error	Unparseable
`mcp-alchemy`	PostgreSQL	48	18	23	0	1
`mcp-alchemy`	DuckDB	54	9	27	0	0
`mcp-alchemy`	MySQL	58	9	23	0	0
`anthropic-pg`	PostgreSQL	52	13	0	16	9
`dbhub`	PostgreSQL	55	12	1	16	6
`dbhub`	MySQL	59	7	0	18	6

Across both scale factors, 149 attempts end at the call limit. Another 101 end with HTTP 400 from OpenRouter, and two end with HTTP 502. All 101 HTTP 400 errors occur on the same six questions: `tb_c2_15`, `tb_c5_04`, and `tb_c5_09` through `tb_c5_12`. These queries return rows rather than a small aggregate. Two return one row per customer, one returns one row per part, and the other three return every distinct customer that satisfies a broad price condition. None uses `LIMIT`. Their SF1 gold results contain about 100,000 to 200,000 rows. Within the Node.js server cells, these questions account for 108 attempts, of which 101 end with HTTP 400. None of the other 24 questions produces this error. The recorded exception comes from OpenRouter's chat-completion endpoint, so the terminal failure occurs at the model API boundary rather than in the DBMS. This pattern is consistent with a large tool result being included in a later model request. However, it does not identify which OpenRouter request-size or validation rule caused the rejection.

This is a coupling-level large-result failure. It appears when the high-cardinality workload, server response, agent loop, and model endpoint are combined. It does not show that MCP or the DBMS fails in isolation. Pagination, aggregation, sampling, or an out-of-band result reference would reduce the amount of data carried into the model request. A controlled payload sweep would be needed to measure the exact rejection threshold.

5.3.3 Scale-factor robustness

Table 5.9 compares the two Track B sweeps. Five cells change by at most 13.4 percentage points. One cell changes much more: `dbhub`×PostgreSQL falls from 61.1% at SF1 to 17.8% at SF3.

The aggregate completion rate falls from 60.4% at SF1 to 53.9% at SF3. This average is misleading because it is driven by one cell. When `dbhub`×PostgreSQL is removed, the other five cells increase slightly from 60.2% to 61.1%. The main scale result is therefore a localized failure, not a general decline.

Table 5.9: Verifier-confirmed completion at SF1 and SF3. Each rate is based on 90 attempts.

Server	Backend	SF1	SF3	Change (pp)
`dbhub`	PostgreSQL	0.611	0.178	−43.3
`dbhub`	MySQL	0.656	0.578	−7.8
`anthropic-pg`	PostgreSQL	0.578	0.511	−6.7
`mcp-alchemy`	DuckDB	0.600	0.600	0.0
`mcp-alchemy`	MySQL	0.644	0.700	+5.6
`mcp-alchemy`	PostgreSQL	0.533	0.667	+13.4

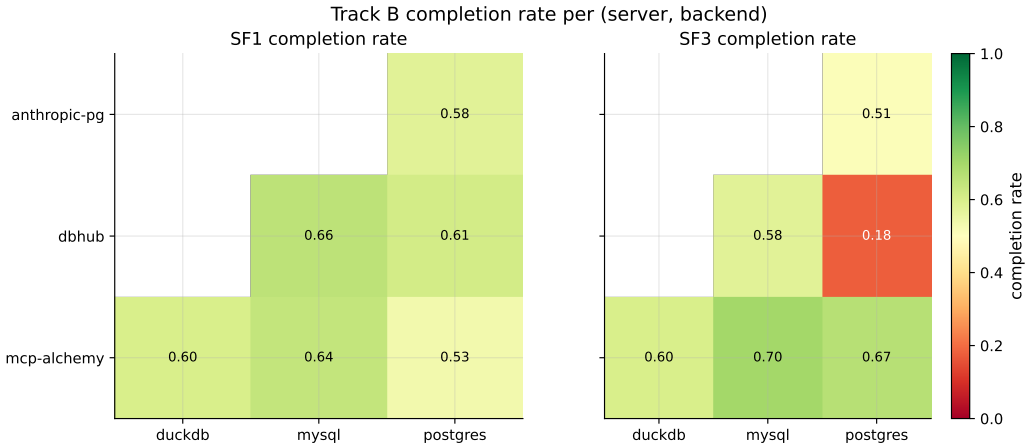


Figure 5.1: Verifier-confirmed completion for all six cells at SF1 and SF3. The main change is the `dbhub`×PostgreSQL cell at SF3.

The `dbhub`×PostgreSQL completion drop

The outcome counts show where the change occurs. At SF1, `dbhub`×PostgreSQL records 55 correct answers, 12 incorrect answers, 16 agent errors, 6 unparseable submissions, and 1 call-limit failure. At SF3, it records 16 correct answers, 49 incorrect answers, 18 agent errors, and 7 unparseable submissions. Thus, 37 additional incorrect answers account for most of the completion loss.

The HTTP 400 errors remain important, but they do not explain the 43.3-point drop. Their count increases only from 16 at SF1 to 18 at SF3. The result instead shows a scale-sensitive correctness regression in this server–backend coupling. The experiment does not isolate whether the regression comes from backend results, server response handling, or the agent's use of those results. A query-level replay would be needed to separate these causes.

5.3.4 Tool surface and tool-call volume

Table 5.10 reports tool-call counts at both scale factors. The ordering changes little with scale. The four-tool `mcp-alchemy` couplings have the three highest means at both SF1 and SF3. The median is unchanged for every coupling. Across all attempts, the mean decreases slightly from 3.79 calls at SF1 to 3.64 at SF3. Thus, there is no consistent increase in tool-call volume with scale in the tested range. The `dbhub`×PostgreSQL completion drop at SF3 also occurs without an increase in its mean tool-call count.

Table 5.10: Tool-call counts per Track B attempt at SF1 and SF3.

			SF1			SF3		
Server	**Backend**	**Tools**	**Mean**	**Median**	**Max.**	**Mean**	**Median**	**Max.**
`anthropic-pg`	PostgreSQL	2	1.24	1	9	1.10	1	6
`dbhub`	MySQL	2	2.47	1	9	2.57	1	7
`dbhub`	PostgreSQL	2	3.96	4	11	3.77	4	12
`mcp-alchemy`	MySQL	4	4.80	3	15	4.81	3	24
`mcp-alchemy`	PostgreSQL	4	5.07	3	24	4.77	3	11
`mcp-alchemy`	DuckDB	4	5.23	3	49	4.84	3	11

The wider surface lets the agent inspect tables and schema before it executes SQL. That can help grounding, but it also consumes loop iterations. Across both scale factors, 148 of the 540 `mcp-alchemy` attempts reach the call limit. The other three cells produce one

such failure in total. The result shows a clear association between this tool design and call-budget use. It does not show that four tools are always worse. A matched ablation would be needed to separate tool count from descriptions, runtime, and response format.

The loop is limited to ten model-and-tool iterations, not ten individual tool calls. One model turn can issue several tools. This is why some traces record more than ten tool calls before ending at the loop limit.

5.3.5 Cost and token use

An attempt is successful when the agent returns the same data as the reference result. The reference result is produced by running the hand-written SQL for that question on the same backend and scale factor. Any other outcome is unsuccessful.

Of the 1,080 attempts, 617 (57.1%) were successful and 463 (42.9%) were unsuccessful. In 298 unsuccessful attempts, the verifier received no answer that it could check. Of these, 149 reached the call limit, 101 ended with HTTP 400, 46 produced no readable answer, and 2 ended with HTTP 502. In the other 165 attempts, the agent returned an answer, but the data was wrong. Among these, 87 were missing at least one expected row, 72 returned the wrong number of rows, and 6 returned the wrong number of values per row.

The unsuccessful attempts consumed $2.512 of the $2.876 total cost, or 87.4%. On average, an unsuccessful attempt cost about 9.2 times more than a successful attempt. Successful attempts stop when the agent calls `submit_answer`. Many unsuccessful attempts continue for additional turns before reaching the call limit or an API error. They therefore consume more model tokens on average.

At SF1, cost per correct answer ranges from USD 0.00293 to USD 0.00622. At SF3, it ranges from USD 0.00270 to USD 0.01965. Table 5.11 reports the SF3 values. The high value for `dbhub`×PostgreSQL follows from its low completion rate.

Table 5.11: Cost per correct answer at SF3.

Server	Backend	USD/correct
`mcp-alchemy`	PostgreSQL	0.00270
`mcp-alchemy`	MySQL	0.00298
`mcp-alchemy`	DuckDB	0.00316
`dbhub`	MySQL	0.00783
`anthropic-pg`	PostgreSQL	0.00828
`dbhub`	PostgreSQL	0.01965

The harness records total input and output tokens per attempt. It does not separate discovery tokens, result tokens, and fixed prompt tokens. Therefore, this chapter reports input-token ratios as proxies. It does not report them as the calibrated token-inflation metric ϕ. This distinction prevents a convenient but incomplete ratio from being treated as a final protocol metric.

5.4 Workload complexity and verifier results

Track B also tests whether SQLyzr complexity transfers to an agentic TPC-H setting. C2 contains simpler constructs. C5 contains nested subqueries, CASE expressions, or both. The experiment uses the two endpoints rather than all six SQLyzr categories. It can therefore test an endpoint difference, but it cannot establish a monotone trend across C1 to C6.

5.4.1 C2 and C5 completion

C2 has a higher SF1 completion rate in all six cells. Table 5.12 shows the result. The within-cell difference ranges from 8.9 to 17.8 percentage points.

Table 5.12: C2 and C5 completion at SF1. Each category contains 45 attempts per cell.

Server	Backend	C2	C5	Difference (pp)
`anthropic-pg`	PostgreSQL	30/45 (66.7%)	22/45 (48.9%)	17.8
`mcp-alchemy`	MySQL	33/45 (73.3%)	25/45 (55.6%)	17.7
`dbhub`	MySQL	33/45 (73.3%)	26/45 (57.8%)	15.5
`mcp-alchemy`	DuckDB	30/45 (66.7%)	24/45 (53.3%)	13.4
`dbhub`	PostgreSQL	30/45 (66.7%)	25/45 (55.6%)	11.1
`mcp-alchemy`	PostgreSQL	26/45 (57.8%)	22/45 (48.9%)	8.9

Across all cells, C2 completes 182 of 270 attempts and C5 completes 144 of 270. The aggregate difference is 14.1 percentage points. A one-tailed sign test over the six independent cells gives $p = 2^{-6} = 0.0156$. A pooled two-proportion test gives $z \approx 3.36$ and $p < 0.001$. The result supports the expected endpoint ordering from SQLyzr (Abedini, 2026).

The result is not a full replication of the SQLyzr benchmark. The schema, model, and execution process are different. The queries were written for TPC-H and checked with Catter. The finding is narrower: C2 remains easier than C5 across every measured coupling in this agent-present workload.

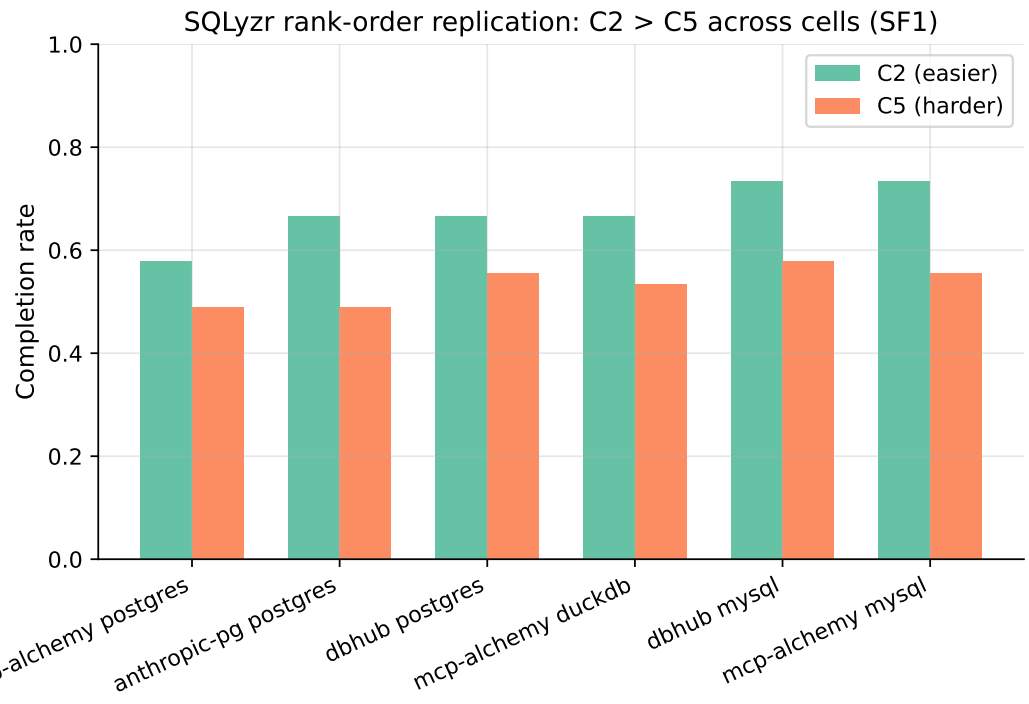


Figure 5.2: C2 and C5 completion at SF1. All six cells show the same endpoint ordering.

5.4.2 Interaction cost by category

C5 also uses more input context in five of the six SF1 cells. Table 5.13 reports the mean cumulative input-token ratio between C5 and C2. A value above one means that C5 uses more input tokens on average.

Table 5.13: Mean cumulative input-token ratio, C5 divided by C2, at SF1. This is a proxy for context growth, not the calibrated IM-8 metric.

Server	Backend	C5/C2 input-token ratio
`dbhub`	PostgreSQL	2.34
`anthropic-pg`	PostgreSQL	1.87
`mcp-alchemy`	DuckDB	1.86
`mcp-alchemy`	MySQL	1.79
`mcp-alchemy`	PostgreSQL	1.41
`dbhub`	MySQL	0.88

The mean is sensitive to long failed attempts and large tool responses. For this reason, the ratios are descriptive. The `dbhub`×PostgreSQL value is the largest, which is consistent with the same cell's SF3 payload failure. The `dbhub`×MySQL value is below one. The trace data do not isolate whether this difference comes from result size, driver behaviour, or the agent path, so no single cause is assigned.

5.4.3 Verifier diagnostics

The verifier produces a diagnostic string for every attempt. Table 5.14 groups those strings into success, structural failure, and content mismatch. The classes cover all 1,080 attempts.

Table 5.14: Verifier diagnostics across both Track B scale factors.

Class	Count	Main diagnostics
Success	617 (57.1%)	590 multiset matches and 27 ordered row-by-row matches
Structural failure	298 (27.6%)	149 call-limit failures, 101 HTTP 400 errors, 46 missing submissions, and 2 HTTP 502 errors
Content mismatch	165 (15.3%)	87 unmatched gold rows, 72 row-count mismatches, and 6 row-width mismatches

This separation matters. A call-limit failure needs a different fix from a wrong value. An HTTP 400 after a large tool result also needs a different fix from invalid SQL. A single pass-or-fail value would hide these differences.

The analyzer also flags possible shape drift, such as a scalar returned as a one-element list or a list encoded as a string. These flags overlap with the verifier outcome and are not treated as another terminal class in Table 5.14. Instead, the verifier records the final answer type and the matching diagnostic. This keeps value errors separate from representation errors.

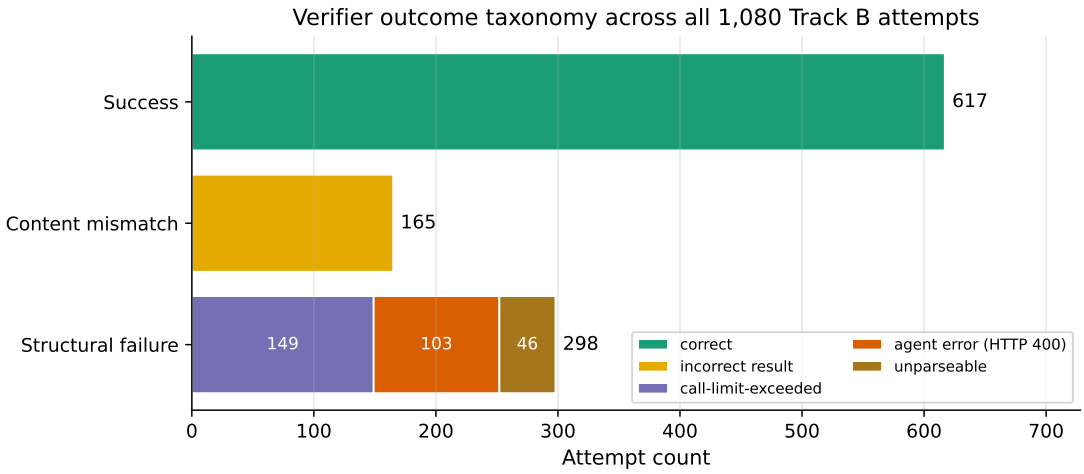


Figure 5.3: Verifier diagnostic classes across all 1,080 Track B attempts.

5.4.4 Per-query difficulty

The category averages hide large differences between individual questions. Eight questions receive no correct answer across their 36 attempts. Six are C5 questions and two are C2 questions. One additional C2 question completes in only 14.8% of attempts. The remaining questions range from 51.9% to 92.6%.

This result limits the category claim. C2 is easier on average, but category does not determine the outcome of every query. Result cardinality and response shape also matter. The full question and gold-SQL list in Appendix A allows readers to inspect these cases directly.

Table 5.15: Queries at the lower end of the pooled completion distribution. Each query has 36 attempts across six cells, two scale factors, and three repetitions.

Query	Category	Completion
`tb_c2_10`, `tb_c2_15`	C2	0.0%
`tb_c5_04`, `tb_c5_05`	C5	0.0%
`tb_c5_09`–`tb_c5_12`	C5	0.0%
`tb_c2_12`	C2	14.8%
11 middle-band queries	C2/C5	51.9–77.8%
10 easy-band queries	Mainly C2	81.5–92.6%

5.5 Cross-track analysis

Track A measures protocol cost without an agent. Track B measures verified completion with an agent. RQ3 asks whether the server ordering from Track A predicts the ordering from Track B. The comparison is made at the server level because the three servers support different backend sets.

5.5.1 Rank-order comparison

Table 5.16 joins the two tracks. Track A uses the complete Π definition, including teardown and the timing residual. Track B uses mean completion across the backends available for each server.

Table 5.16: Server-level join of Track A median Π and Track B mean completion.

Scale factor	Server	Median Π	Mean completion	Mean R proxy
SF1	`mcp-alchemy`	1.214	0.593	4.03
SF1	`dbhub`	1.662	0.633	2.21
SF1	`anthropic-pg`	4.055	0.578	0.24
SF3	`mcp-alchemy`	1.073	0.656	3.81
SF3	`dbhub`	1.160	0.378	2.17
SF3	`anthropic-pg`	1.972	0.511	0.10

The rank correlation between Π and completion is $r_s = -0.5$ at both scale factors. With only three servers, the exact two-sided p-value is 1.0. The result is descriptive and cannot support a population-level correlation claim.

The rankings do not match. At SF1, `dbhub` has higher protocol tax than `mcp-alchemy` but also higher completion. At SF3, `dbhub` has the lowest completion because one backend cell collapses. Meanwhile, `anthropic-pg` has the largest complete Π at both scales because of its shutdown behaviour, but its completion remains in the middle of the SF3 ranking.

An earlier six-phase analysis excluded teardown and produced a sign change in the correlation. That sign change disappears when the metric is calculated as defined in Chapter 3. The corrected result is $r_s = -0.5$ at both scales. This correction does not make the correlation significant. It shows that the conclusion is sensitive to the accounting boundary and that three servers are not enough for RQ3.

5.5.2 Retry proxy

The executed harness does not populate the direct retry-time field. This chapter therefore uses $n_{\text{tool calls}} - 1$ as a proxy for IM-7. The proxy includes schema exploration and repeated calls, so it is broader than a strict retry count.

The proxy follows tool-surface design more closely than Π. `mcp-alchemy` is near four extra calls per attempt at both scales. `anthropic-pg` is near zero. `dbhub` lies between them. A server can therefore have a low transport cost but still consume many agent turns. Protocol time and agent interaction cost should be reported separately.

5.5.3 Statistical detectability

At $n = 3$, no Spearman ordering can reach a two-sided significance level of 0.05. The exact permutation distribution is too coarse. The benchmark specification identifies six servers as the minimum useful expansion for this rank test, where $|r_s| > 0.886$ would be required. Until that expansion is run, RQ3 should be reported as inconclusive.

5.6 Scope and limitations

The experiment covers the main timing and agent-outcome questions, but it does not execute every metric or workload variant defined in Chapter 3. Table 5.17 lists the limits that affect interpretation.

The single model and single wording are the main Track B limits. The reported rates describe one fixed agent configuration. More wordings are needed to measure robustness to phrasing, and more models are needed to test model-family transfer.

Table 5.17: Main limits of the reported experiments.

Aspect	**Executed scope**	**Interpretation**
Track B model	One DeepSeek v4 Flash configuration	Completion and failure rates may change with another model
Question wording	One wording, repeated three times	Repetitions measure consistency, not paraphrase robustness
SQLyzr coverage	C2 and C5 only	Supports an endpoint comparison, not a full C1-to-C6 trend
Cross-track sample	Three servers	Rank correlation is descriptive and cannot be significant
Execution order	Sequential on one host	No concurrent-load or randomized-order conclusion is supported
Track A cold state	Fresh runner and connection state	Does not guarantee an empty DBMS buffer cache
Database reference	Separate native connection before MCP call	Does not measure the exact in-server SQL interval
Tracing calibration	No separate untraced sweep	Closure is checked, but observer overhead is not measured directly

The three-server sample is the main RQ3 limit. It cannot support a significant Spearman result. The Track A and Track B server averages also cover different backend sets. `mcp-alchemy` is averaged over three backends, `dbhub` over two, and `anthropic-pg` over one. This is acceptable for a pilot server-level comparison, but it is not a fully balanced rank test.

The measurement design has two additional limits. First, T_{DB} comes from a native reference call rather than the SQL call inside the server. Second, the traced campaign has no separate untraced calibration. These choices do not invalidate the phase comparisons, but they limit claims about exact in-server execution time and sub-millisecond observer overhead.

5.6.1 Metric coverage

Table 5.18 maps the sixteen metrics from Chapter 3 to the executed data. The timing metrics are measured directly. Several byte, resource, and token metrics remain proxies or were not aggregated.

Table 5.18: Coverage of the Implementation and Coupling metrics.

ID	Metric	Status	Result or limitation
IM-1	Token inflation ϕ	Proxy	Total input/output token ratios only; no envelope decomposition
IM-2	MCP payload size B_{MCP}	Recorded	Byte fields exist in traces but were not aggregated per cell
IM-3	Parse latency T_{parse}	Measured	Reported in Track A phase traces
IM-4	Discovery latency T_{disc}	Measured	Reported in Track A phase traces
IM-5	Lifecycle overhead T_{life}	Measured	Major Track A result
IM-6	Cold-to-warm delta δ_{CW}	Measured	Uses runner and connection reuse, not a guaranteed cold DBMS cache
IM-7	Retry-loop cost R	Proxy	Uses $n_{\mathrm{tool\ calls}} - 1$ because direct retry time is absent
IM-8	Context decay γ	Proxy	Uses cumulative input-token ratios
IM-9	Discovery tokens τ_{disc}	Not measured	No tokenizer pass over Track A discovery payloads
IM-10	Lifecycle calls n_{life}	Measured	Equal to one in all executed Track A traces
CM-1	Serialization throughput S_{ser}	Not valid	Span covers only outer adapter encoding
CM-2	Payload expansion ρ	Not measured	Native response-byte baseline is absent
CM-3	Peak memory M_{peak}	Partial	Available for some cells but not complete across servers
CM-4	CPU threshold $\sigma^{*}_{\mathrm{CPU}}$	Not measured	No CPU-saturation sweep
CM-5	Connection handover T_{pool}	Not separated	Pool acquisition is included in lifecycle
CM-6	Back-pressure β	Qualitative	Large-result failure is observed, but no dedicated load sweep was run

The coverage gap is concentrated in byte accounting, resource saturation, and calibrated token analysis. The available traces are sufficient for the timing, completion, tool-call, failure-mode, and cost results reported in this chapter. They are not sufficient for a final claim about native-to-MCP payload expansion or serialization throughput.

5.7 Summary of findings

Table 5.19 summarizes the main findings. It separates measured results from descriptive or underpowered comparisons.

Table 5.19: Main experimental findings.

No.	Finding	Evidence
1	Protocol Tax varies strongly across couplings	Cell-median Π ranges from 0.0031 to 7.32
2	Fixed protocol cost is amortized as data scale grows	Π falls in 119 of 120 adjacent scale comparisons
3	Lifecycle and transport explain most non-embedded overhead	Representative phase traces; five-second `anthropic-pg` shutdown
4	Faster SQL can expose middleware cost	Q6 SF1 indexed Π rises from 1.94 to 3.03 cold and 2.40 to 4.70 warm
5	SF1 completion alone does not separate the six cells	Rates range from 53.3% to 65.6% with overlapping intervals
6	One coupling fails under larger results	`dbhub`×PostgreSQL falls 43.3 percentage points at SF3
7	Tool design changes agent behaviour	Four-tool cells produce 148 of 149 call-limit failures
8	C2 is easier than C5 in every cell	Aggregate difference 14.1 points; sign-test $p = 0.0156$
9	Failed attempts dominate model spend	87.4% of USD 2.876 is spent on unsuccessful attempts
10	RQ3 remains inconclusive	$r_s = -0.5$ at SF1 and SF3; exact $p = 1.0$, $n = 3$

5.7.1 Answers to the research questions

RQ1: When does protocol cost become important? Protocol cost is already comparable to database time in five of the six couplings. It dominates many SF1 cells and becomes less important as scale grows. The largest cell median is 7.32, so the experiment does not show orders-of-magnitude dominance over database time. DuckDB is different because it runs in process. The index experiment confirms the same principle from the other direction: when SQL becomes faster, fixed middleware cost takes a larger share.

RQ2: Which implementation choices explain the difference? Lifecycle and transport are the main measured phases. Process shutdown is especially important for

`anthropic-pg`. In Track B, tool-surface design changes how quickly the agent reaches its loop limit. Large response payloads also create a context-related failure in one server–backend coupling. These results support separate Implementation and Coupling metric tiers.

RQ3: Does lower Protocol Tax predict better agent completion? The current data do not establish such a relationship. The rank correlation is -0.5 at both scale factors and is not testable at $n = 3$. More importantly, the cell-level results show that protocol speed and agent success can move independently. DuckDB has the lowest Π but only middle-range completion, while `dbhub`×PostgreSQL has a moderate Track A ratio and the largest Track B failure at SF3.

The chapter therefore supports a layered conclusion. Protocol cost is measurable and often large. However, it is only one part of agentic database performance. Tool design, result shape, context growth, and correctness must be measured with it. A single end-to-end latency or completion score cannot identify which layer needs to change.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis studied the cost of placing MCP between an agent and a DBMS. It defined Protocol Tax as all measured interaction time outside the native database reference. The measurement model divides this cost into seven protocol phases and a residual. MADBench then measures these phases under controlled SQL workloads and compares them with agent completion under natural-language workloads.

The experiments contain 5,378 Track A traces and 1,080 Track B attempts. Track A covers six server and backend combinations, five TPC-H queries, three scale factors, and separate cold and warm runner states. Track B uses the same six combinations at SF1 and SF3. It contains 30 natural-language questions with three repetitions per question. Together, the two tracks show that protocol cost and agent success describe different parts of the system.

6.1.1 Answers to the research questions

RQ1: Scale transition. Protocol cost is already comparable to database execution in five of the six tested couplings. Their median Π values are above one, while the in-process DuckDB coupling has a median of 0.019. Across all Track A cells, the median Π ranges from 0.0031 to 7.32. The upper value shows substantial overhead, but it does not show orders-of-magnitude dominance over database time.

The expected upward transition with scale does not occur in the measured workload. Instead, Π falls in 119 of the 120 adjacent scale comparisons. The five client-server couplings remain above one at the median through SF10, but their ratios move closer to one. DuckDB remains below one. This happens because the selected TPC-H queries have fixed result cardinalities. Database work grows with the scale factor, while much of the protocol path remains fixed. The index experiment confirms the same relationship from the other direction. When an index makes Q6 faster at SF1, fixed middleware cost becomes a larger share of the interaction. These results answer RQ1 within the tested range: Protocol Tax is often important, but the study does not observe a scale factor at which it begins to grow relative to database execution.

RQ2: Attribution. The phase traces identify lifecycle, transport, and teardown as the main measured sources of non-embedded overhead. Their importance depends on the server and backend combination. For example, the `anthropic-pg` adapter waits about 5 s before terminating its child process. Including this teardown interval raises its median PostgreSQL Π to 1.874. Excluding it would change the server ordering.

Track B shows a different set of implementation effects. The four-tool `mcp-alchemy` surface accounts for 148 of the 149 call-limit failures. The two-tool servers use fewer calls but produce most of the HTTP 400 agent errors. These errors follow large tool responses and are consistent with a payload or context limit in the next model request. However, the OpenRouter response body was not recorded, so the exact provider-side cause is unknown. The observed behaviour therefore belongs to the complete coupling of server, backend, response format, agent loop, and model service. It cannot be assigned to the programming language or MCP alone.

These results support the distinction between Implementation Metrics and Coupling Metrics. Tool count, lifecycle policy, and response design provide useful implementation diagnostics. Other measurements, such as large-result behaviour, depend on several layers and are more useful for comparing complete stacks. Some planned byte and resource metrics were not measured across the full grid. RQ2 is therefore answered for the timing and agent-behaviour measurements reported here, not for every metric in the original specification.

RQ3: Predictive validity. Lower Protocol Tax does not guarantee higher agent completion. DuckDB has the lowest Π but only a middle-range completion rate. At the server level, the Spearman correlation between Π and completion is $r_s = -0.5$ at both SF1 and

SF3. The exact two-sided p-value is 1.0. Only three servers appear in both tracks, so this analysis is descriptive and cannot support a general correlation claim.

The cell-level results still show why both tracks are needed. The six SF1 completion rates range from 53.3% to 65.6%, and their Wilson intervals overlap. At SF3, `dbhub` with PostgreSQL falls from 61.1% to 17.8%, while the other five cells remain within 13.4 percentage points of their SF1 values. This localized failure does not follow the general Track A scale trend. Thus, controlled protocol measurements cannot replace agent-present evaluation. RQ3 remains inconclusive at the server level, but the experiments show that protocol speed and agent completion can move independently.

Query complexity also changes agent completion. At SF1, C2 completes 182 of 270 attempts, while C5 completes 144 of 270. The 14.1-point difference appears in all six cells. Failed attempts also consume 87.4% of the total model spend because many continue until a tool-call or context limit. Completion should therefore be reported with workload category and interaction cost.

RQ4: Methodology. MADBench provides one measurement method for these systems. Track A uses OpenTelemetry spans with explicit boundaries for discovery, lifecycle, handover, database execution, serialization, parsing, transport, and teardown. Track B uses event traces that record tool calls, token use, cost, submitted answers, and terminal outcomes. A deterministic verifier separates correct answers, content mismatches, unparseable submissions, call-limit failures, and agent errors. It also records shape drift without treating it as a separate terminal class.

The method combines two forms of control. Track A removes model variance and measures the protocol path under fixed SQL. Track B keeps the agent and measures the effects that appear only during iterative tool use. The shared server, backend, scale-factor, and workload identifiers allow results from the two tracks to be compared. The framework also separates metrics that mainly reflect server decisions from those that describe the complete coupling.

The executed study also shows the limits of the method. Database time comes from a separate native reference call rather than the SQL call inside the MCP server. The campaign has no separate untraced calibration run. Several byte, CPU, memory, and serialization measurements are incomplete or unavailable. These limits are reported with the results and restrict the corresponding claims.

6.1.2 Contributions

The thesis makes four contributions.

1. It defines Protocol Tax through a seven-phase timing model with explicit measurement boundaries. The model separates database execution from discovery, lifecycle, handover, serialization, parsing, transport, and teardown.

2. It introduces a two-tier attribution framework. Implementation Metrics identify effects tied mainly to server design. Coupling Metrics describe effects that arise from the combined server, driver, runtime, and DBMS stack.

3. It develops a two-track experimental design. Track A provides controlled protocol measurements without an agent. Track B measures completion, tool use, context growth, and failure behaviour with an agent present.

4. It implements these ideas in MADBench. The benchmark includes the executed workloads, trace pipeline, correctness verifier, run manifests, derived tables, and analysis scripts. The full workload and gold SQL are also provided in Appendix A.

6.2 Future work

6.2.1 Broader agent evaluation

Track B uses one model, one wording per question, and two SQLyzr categories. The three repetitions measure consistency for that fixed setup. They do not measure robustness to paraphrasing or transfer across models. A larger study should add the remaining C1, C3, C4, and C6 categories. It should also run several phrasings of each question and repeat the workload with other models. This would separate query difficulty from wording and model effects.

The cross-track comparison also needs more servers. The current three-server sample cannot support a significant rank test. Expanding the study to the planned six servers would provide a more useful comparison and add other implementation lineages. The analysis should keep the backend set balanced across servers so that a server-level rank does not combine different backend populations.

6.2.2 Large-result failures

The SF3 results motivate a targeted large-result experiment. The current data associate HTTP 400 errors with large tool responses, but they do not contain the rejected OpenRouter response body or the exact request size. A follow-up run should record both values and preserve partial agent state when a model request fails. It should then repeat the affected questions with capped, paginated, and referenced results. This would test whether limiting the tool payload removes the failure.

Track B can be extended to SF10 after this mechanism is instrumented. A third scale factor alone would show whether the failure persists, but it would not identify the cause. The controlled payload variants are needed to separate server response design from provider or model limits.

6.2.3 Measurement coverage

Several measurements require additional instrumentation. Teardown should be divided into server cleanup, pipe draining, and forced termination where the runtime exposes these events. This change requires new spans and new runs. The agent runner should also retain completed turns when a later request fails. This would prevent partial tool and token histories from being replaced by zeros.

An untraced calibration sweep is needed to measure observer overhead directly. Instrumentation inside each MCP server would also provide the actual in-server database interval instead of a separate native reference. Complete byte accounting would support the Payload Expansion Ratio, while consistent CPU and memory sampling would support the remaining Coupling Metrics. These additions would strengthen attribution without changing the workload.

The workload should also include at least one query whose result cardinality grows with scale. The five current Track A queries keep output cardinality fixed. They can measure how fixed protocol cost is amortized, but they cannot test the growing-result condition in Lemma 3.1. An uncapped result query would provide that test.

6.2.4 Other protocol designs

The timing model can also be tested outside MCP. A long-lived JSON-RPC service would preserve text encoding while removing per-session process startup. A streaming binary

implementation would change the encoding and result-transfer model. Comparing these designs with the same workload would help separate process cost from transport and encoding cost. Such a study would test whether the seven phase boundaries remain useful when the protocol changes.

6.3 Final remarks

The experiments show that MCP-mediated database access cannot be evaluated with one latency or completion value. Protocol cost is often comparable to database execution, but its relative importance falls as database work grows in the tested workload. Agent completion follows a different pattern and can fail because of tool use or large result payloads. MADBench makes these effects visible through separate tracks and explicit timing boundaries. The next step is to extend the server and model coverage while completing the missing payload and resource measurements.

References

Abedini, S. (2026). SQLyzr: A comprehensive benchmark and framework for evaluating text-to-SQL systems. Master's thesis, University of Waterloo.

Akillioglu, K., Chakraborty, A., Voruganti, S., and Özsu, M. T. (2025). Research challenges in relational database management systems for LLM queries. In *Proc. 6th Intl. Wkshp on Applied AI for Database Syst. and Appl.* Co-located with VLDB 2025.

Anthropic (2024). MCP PostgreSQL server (archived). https://github.com/modelcontextprotocol/servers-archived/tree/main/src/postgres. Archived; no longer maintained.

Anthropic (2025). Model context protocol specification. https://modelcontextprotocol.io/specification/2025-03-26/basic. Version 2025-03-26.

Axiom (2025). Designing MCP servers for wide schemas and large result sets. https://axiom.co/blog/designing-mcp-servers-for-wide-events. Accessed: 2026-03-10.

Bentham, M. (2024). SqlAugur: MCP server for SQL Server database analysis. https://github.com/mbentham/SqlAugur. 31 tools organized into dynamic toolsets. Accessed: 2026-03-10.

Bian, S., Yan, M., Jayarajan, A., Pekhimenko, G., and Venkataraman, S. (2025). What limits agentic systems efficiency? *CoRR*, abs/2510.16276.

Bloomberry (2026). I analyzed 1400 MCP servers: Here's what I learned. https://bloomberry.com/blog/we-analyzed-1400-mcp-servers-heres-what-we-learned/. Accessed: 2026-03-10.

Boncz, P. A., Zukowski, M., and Nes, N. (2005). MonetDB/X100: Hyper-pipelining query execution. In *Proc. 2nd Biennial Conf. on Innovative Data Systems Research*, pages 225–237.

Borla, B. (2024). MCP server for MySQL. https://github.com/benborla/mcp-server-mysql. Accessed: 2026-03-10.

Bytebase (2025). DBHub: Database MCP server. https://github.com/bytebase/dbhub. Accessed: 2026-03-10.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387.

Datadog Security Labs (2025). MCP vulnerability case study: SQL injection in the Postgres MCP server. https://securitylabs.datadoghq.com/articles/mcp-vulnerability-case-study-SQL-injection-in-the-postgresql-mcp-server/. Accessed: 2026-03-10.

Dean, J. and Barroso, L. A. (2013). The tail at scale. *Commun. ACM*, 56(2):74–80.

designcomputer (2024). MySQL MCP server. https://github.com/designcomputer/mysql_mcp_server. Accessed: 2026-03-10.

Ding, Z., Zhu, M., and Liu, Y. (2025). Network and systems performance characterization of MCP-enabled LLM agents. *CoRR*, abs/2511.07426.

Dixon, W. J. and Mood, A. M. (1946). The statistical sign test. *J. American Statistical Assoc.*, 41(236):557–566.

Dong, J.-K., Huang, I.-W., Wu, C.-T., and Tsai, Y.-T. (2026). ETOM: A five-level benchmark for evaluating tool orchestration within the MCP ecosystem. In *Findings of the Assoc. for Computational Linguistics: EACL 2026*, pages 1453–1488.

DuckDB Foundation (2025). DuckDB documentation. https://duckdb.org/docs/stable/. Accessed: 2026-03-10.

Efron, B. (1979). Bootstrap methods: Another look at the jackknife. *Annals of Statistics*, 7(1):1–26.

Ehtesham, A., Singh, A., Gupta, G. K., and Kumar, S. (2025). A survey of agent interoperability protocols: Model context protocol (MCP), agent communication protocol (ACP), agent-to-agent protocol (A2A), and agent network protocol (ANP). *CoRR*, abs/2505.02279.

Fan, S., Ding, X., Zhang, L., and Mo, L. (2025). MCPToolBench++: A large scale AI agent model context protocol MCP tool use benchmark. *CoRR*, abs/2508.07575.

Faucet (2026). Your indexes matter 10x more than your database and MySQL knew that all along. https://faucetdb.ai/blog/mcp-database-benchmark/. Faucet Blog. Industrial MCP benchmark across PostgreSQL, MySQL, SQL Server, SQLite. Published 2026-03-09. Accessed: 2026-08-04.

Fernandez, R. C., Elmore, A. J., Franklin, M. J., Krishnan, S., and Tan, C. (2023). How large language models will disrupt data management. *Proc. VLDB Endowment*, 16(11):3302–3309.

Gao, X., Xie, S., Zhai, J., Ma, S., and Shen, C. (2025). MCP-RADAR: A multi-dimensional benchmark for evaluating tool use capabilities in large language models. *CoRR*, abs/2505.16700.

General Analysis (2025). Supabase MCP can leak your entire SQL database. https://www.generalanalysis.com/blog/supabase-mcp-blog. Accessed: 2026-03-10.

Google Cloud (2025). MCP Toolbox for Databases. https://github.com/googleapis/mcp-toolbox. Formerly published as googleapis/genai-toolbox. Accessed: 2026-08-04.

Graefe, G. (1993). Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170.

gRPC Authors (2016). Mobile benchmarks. https://grpc.io/blog/mobile-benchmarks/. Published 2016-07-26. Accessed: 2026-08-04.

Guo, Z., Xu, B., Zhu, C., Hong, W., Wang, X., and Mao, Z. (2026). MCP-AgentBench: Evaluating real-world language agent performance with MCP-mediated tools. In *Proc. 40th AAAI Conf. on Artificial Intelligence*, pages 30888–30896.

Hartenfeller, P. (2025). Demystifying the SQLcl MCP server. https://hartenfeller.dev/blog/demystifying-sqlcl-mcp-server. Accessed: 2026-03-10.

Hong, Z., Chen, H., Yuan, Z., Zhang, Q., Zhuang, L., Liao, Q., Huang, F., Song, Y., and Huang, X. (2026). ErrorLLM: Modeling SQL errors for text-to-SQL refinement. *CoRR*, abs/2603.03742.

Hong, Z., Yuan, Z., Zhang, Q., Chen, H., Dong, J., Huang, F., and Huang, X. (2025). Next-generation database interfaces: A survey of LLM-based text-to-SQL. *IEEE Trans. Knowl. and Data Eng.*, 37(12):7328–7345.

Hou, X., Zhao, Y., Wang, S., and Wang, H. (2026). Model context protocol (MCP): Landscape, security threats, and future research directions. *ACM Trans. Softw. Eng. and Meth.* Just Accepted.

Huang, J. Y., Damani, M., El-Kurdi, Y., Astudillo, R., and Sun, W. (2025). Latency and token-aware test-time compute. *CoRR*, abs/2509.09864.

Jayanti, M. A. and Han, X. (2026). Enhancing model context protocol (MCP) with context-aware server collaboration. *CoRR*, abs/2601.11595.

Jiang, J., Shen, S., Xie, H., Li, Y., Shen, Y., Huang, D., Qian, B., Wu, Y., Zhang, W., Cui, B., and Chen, P. (2025). SQLGovernor: An LLM-powered SQL toolkit for real world application. *CoRR*, abs/2509.08575.

JSON-RPC Working Group (2010). JSON-RPC 2.0 specification. https://www.jsonrpc.org/specification. Accessed: 2026-03-10.

Kaagaard, R. (2025). mcp-alchemy: MCP server for database access via SQLAlchemy. https://github.com/runekaagaard/mcp-alchemy. Accessed: 2026-03-10.

Kemper, A. and Neumann, T. (2011). HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proc. 27th IEEE Intl. Conf. on Data Engineering*, pages 195–206.

Kersten, T., Leis, V., Kemper, A., Neumann, T., Pavlo, A., and Boncz, P. (2018). Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endowment*, 11(13):2209–2222.

ktanaka101 (2024). MCP server for DuckDB. https://github.com/ktanaka101/mcp-server-duckdb. Accessed: 2026-03-10.

Langdale, G. and Lemire, D. (2019). Parsing gigabytes of JSON per second. *VLDB J.*, 28(6):941–960.

Lei, F., Chen, J., Ye, Y., Cao, R., Shin, D., Su, H., Suo, Z., Gao, H., Hu, W., Yin, P., Zhong, V., Xiong, C., Sun, R., Liu, Q., Wang, S. I., and Yu, T. (2025). Spider 2.0: Evaluating language models on real-world enterprise text-to-SQL workflows. In *Proc. 13th Intl. Conf. on Learning Representations*.

Lemburg, M.-A. (1999). PEP 249 – Python database API specification v2.0. https://peps.python.org/pep-0249/. Last modified 2020.

Li, J., Hui, B., Qu, G., Yang, J., Li, B., Li, B., Wang, B., Qin, B., Geng, R., Huo, N., Zhou, X., Ma, C., Li, G., Chang, K. C., Huang, F., Cheng, R., and Li, Y. (2023). Can LLM already serve as a database interface? a BIg bench for large-scale database grounded text-to-SQLs. In *Advances in Neural Information Processing Syst., Proc. 36th Annual Conf. on Neural Information Processing Syst.*

Liu, C., Pavlenko, A., Interlandi, M., and Haynes, B. (2023). A deep dive into common open formats for analytical dbmss. *Proc. VLDB Endowment*, 16(11):3044–3056.

Liu, S., Biswal, A., Kamsetty, A., Cheng, A., Schroeder, L. G., Patel, L., Cao, S., Mo, X., Stoica, I., Gonzalez, J. E., and Zaharia, M. (2025). Optimizing LLM queries in relational data analytics workloads. In *Proc. 8th Conf. on Machine Learning and Syst.*

Liu, S., Ponnapalli, S., Shankar, S., Zeighami, S., Zhu, A., Agarwal, S., Chen, R., Suwito, S., Yuan, S., Stoica, I., Zaharia, M., Cheung, A., Crooks, N., Gonzalez, J. E., and Parameswaran, A. G. (2026). Supporting our AI overlords: Redesigning data systems to be agent-first. In *Proc. 16th Biennial Conf. on Innovative Data Systems Research*.

Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., Zhang, S., Deng, X., Zeng, A., Du, Z., Zhang, C., Shen, S., Zhang, T., Su, Y., Sun, H., Huang, M., Dong, Y., and Tang, J. (2024). AgentBench: Evaluating LLMs as agents. In *Proc. 12th Intl. Conf. on Learning Representations*.

Luo, Z., Shen, Z., Yang, W., Zhao, Z., Jwalapuram, P., Saha, A., Sahoo, D., Savarese, S., Xiong, C., and Li, J. (2025a). MCP-Universe: Benchmarking large language models with real-world model context protocol servers. *CoRR*, abs/2508.14704.

Luo, Z., Shi, X., Lin, X., and Gao, J. (2025b). Evaluation report on MCP servers. *CoRR*, abs/2504.11094.

Melnik, S., Gubarev, A., Long, J. J., Romer, G., Shivakumar, S., Tolton, M., and Vassilakis, T. (2010). Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endowment*, 3(1):330–339.

Microsoft Corporation (2026). [MS-TDS]: Tabular data stream protocol. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-tds/. Open Specifications, rev. 42.0. Accessed: 2026-08-04.

Mo, G., Zhong, W., Chen, J., Yuan, Q., Chen, X., Lu, Y., Lin, H., He, B., Han, X., and Sun, L. (2025). LiveMCPBench: Can agents navigate an ocean of MCP tools? *CoRR*, abs/2508.01780.

MotherDuck (2025). MotherDuck MCP server for DuckDB. https://github.com/motherduckdb/mcp-server-motherduck. Accessed: 2026-03-10.

Neumann, T. (2011). Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endowment*, 4(9):539–550.

nodejs-security.com (2025). SQL injection and bypassing "read-only" mode in Xata's MCP server. https://www.nodejs-security.com/blog/sql-injection-and-bypassing-read-only-mode-in-xata-mcp-server. Accessed: 2026-03-10.

OpenTelemetry Authors (2025). OpenTelemetry specification. https://opentelemetry.io/docs/specs/otel/. Accessed: 2026-03-10.

Oracle Corporation (2025a). MySQL client/server protocol. https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_PROTOCOL.html. Accessed: 2026-03-10.

Oracle Corporation (2025b). Oracle SQLcl MCP server. https://docs.oracle.com/en/database/oracle/sql-developer-command-line/25.2/sqcug/using-oracle-sqlcl-mcp-server.html. Accessed: 2026-03-10.

Oracle Corporation (2026). JDBC developer's guide. https://docs.oracle.com/en/database/oracle/oracle-database/26/jjdbc/. Oracle AI Database 26ai. Accessed: 2026-08-04.

Page, D. (2026). Lessons learned writing an MCP server for PostgreSQL. https://www.pgedge.com/blog/lessons-learned-writing-an-mcp-server-for-postgresql. pgEdge Inc. Published 2026-02-18. Accessed: 2026-08-04.

Patki, N., Wedge, R., and Veeramachaneni, K. (2016). The synthetic data vault. In *Proc. 3rd IEEE Intl. Conf. on Data Science and Advanced Analytics*, pages 399–410.

PlanetScale (2025). PlanetScale model context protocol (MCP). https://planetscale.com/docs/connect/mcp. Accessed: 2026-03-10.

PostgreSQL Global Development Group (2025). PostgreSQL frontend/backend protocol. https://www.postgresql.org/docs/current/protocol.html. Accessed: 2026-03-10.

Pourreza, M., Li, H., Sun, R., Chung, Y., Talaei, S., Kakkar, G. T., Gan, Y., Saberi, A., Özcan, F., and Arık, S. Ö. (2025). CHASE-SQL: Multi-path reasoning and preference optimized candidate selection in text-to-SQL. In *Proc. 13th Intl. Conf. on Learning Representations*.

Pourreza, M. and Rafiei, D. (2023). DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction. In *Advances in Neural Information Processing Syst., Proc. 36th Annual Conf. on Neural Information Processing Syst.*

Raasveldt, M. and Mühleisen, H. (2019). Duckdb: an embeddable analytical database. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 1981–1984.

Schmidt, T., Leis, V., Boncz, P., and Neumann, T. (2025). Sqlstorm: Taking database benchmarking into the llm era. *Proc. VLDB Endowment*, 18(11):4144–4157.

Shi, J., Xu, B., Liang, J., Xiao, Y., Chen, J., Xie, C., Wang, P., and Wang, W. (2025). Gen-SQL: Efficient text-to-SQL by bridging natural language question and database schema with pseudo-schema. In *Proc. 31st Intl. Conf. on Computational Linguistics*, pages 3794–3807.

Shi, L., Tang, Z., Zhang, N., Zhang, X., and Yang, Z. (2026). A survey on employing large language models for text-to-SQL tasks. *ACM Comput. Surv.*, 58(2):54:1–54:37.

Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure. Technical Report dapper-2010-1, Google, Inc.

Spearman, C. (1904). The proof and measurement of association between two things. *American J. Psychology*, 15(1):72–101.

Sterling, T. (2025). duckdb_mcp: Native DuckDB extension for MCP. https://github.com/teaguesterling/duckdb_mcp. Accessed: 2026-03-10.

Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E. J., O'Neil, P. E., Rasin, A., Tran, N., and Zdonik, S. B. (2005). C-store: A column-oriented DBMS. In *Proc. 31st Intl. Conf. on Very Large Data Bases*, pages 553–564.

Stonebraker, M. and Rowe, L. (1986). The design of POSTGRES. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 340–355.

Talluri, S., Di Donato, G. W., Danelutti, L., Vlaswinkel, K. R., Arnaboldi, M., Delamare, A., Santambrogio, M. D., and Bonetta, D. (2025). Gpjson: High-performance json data processing on gpus. *Proc. VLDB Endowment*, 18(9):3216–3229.

TM Dev Lab (2026). Multi-language MCP server performance benchmark. https://www.tmdevlab.com/mcp-server-performance-benchmark.html. 3.9 million requests tested across Java, Go, Node.js, and Python.

Transaction Processing Performance Council (2022). TPC-H benchmark specification. https://www.tpc.org/tpch/. Version 3.0. Accessed: 2026-08-04.

Trend Micro Research (2025). Why a classic MCP server vulnerability can undermine your entire AI agent. https://www.trendmicro.com/en_us/research/25/f/why-a-classic-mcp-server-vulnerability-can-undermine-your-entire-ai-agent.html. Accessed: 2026-03-10.

Viotti, J. C. and Kinderkhedia, M. (2022). A benchmark of JSON-compatible binary serialization specifications. *CoRR*, abs/2201.03051.

Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., Zhao, W. X., Wei, Z., and Wen, J.-R. (2024). A survey on large language model based autonomous agents. *Front. Comput. Sci.*, 18(6):186345.

Wang, Y., Liu, P., and Yang, X. (2025). LinkAlign: Scalable schema linking for real-world large-scale multi-database text-to-SQL. In *Proc. 2025 Conf. on Empirical Methods in Natural Language Processing*, pages 977–991.

Wang, Z., Yu, G., and Lyu, M. R. (2026). AI-NativeBench: An open-source white-box agentic benchmark suite for AI-native systems. *ACM Trans. Softw. Eng. and Meth.* Just Accepted.

Wilson, E. B. (1927). Probable inference, the law of succession, and statistical inference. *J. American Statistical Assoc.*, 22(158):209–212.

Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., Zhang, M., Wang, J., Jin, S., Zhou, E., Zheng, R., Fan, X., Wang, X., Xiong, L., Zhou, Y., Wang, W., Jiang, C., Zou, Y., Liu, X., Yin, Z., Dou, S., Weng, R., Qin, W., Zheng, Y., Qiu, X., Huang, X., Zhang, Q., and Gui, T. (2025). The rise and potential of large language model based agents: A survey. *Science China Inf. Sci.*, 68(2):121101.

Xu, F. F., Song, Y., Li, B., Tang, Y., Jain, K., Bao, M., Wang, Z. Z., Zhou, X., Guo, Z., Cao, M., Yang, M., Lu, H. Y., Martin, A., Su, Z., Maben, L., Mehta, R., Chi, W., Jang, L., Xie, Y., Zhou, S., and Neubig, G. (2025). TheAgentCompany: Benchmarking LLM agents on consequential real world tasks. In *Advances in Neural Information Processing Syst., Proc. 39th Annual Conf. on Neural Information Processing Syst.*

Yang, H., Zhang, J., He, Z., Zhou, A., and Fung, Y. R. (2025a). MARS-SQL: A multi-agent reinforcement learning framework for text-to-SQL. *CoRR*, abs/2511.01008.

Yang, Y., Gao, C., Wu, D., Chen, Y., Li, Y., and Wang, S. (2025b). MCPSecBench: A systematic security benchmark and playground for testing model context protocols. *CoRR*, abs/2508.13220.

Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., and Radev, D. (2018). Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proc. 2018 Conf. on Empirical Methods in Natural Language Processing*, pages 3911–3921.

Zong, X., Shen, Z., Wang, L., Lan, Y., and Yang, C. (2026). MCP-SafetyBench: A benchmark for safety evaluation of large language models with real-world MCP servers. In *Proc. 14th Intl. Conf. on Learning Representations*.

Appendix A

Workload Queries

This appendix lists the exact natural-language questions and the reference SQL used in Track B. The full pre-registered YAML is at `workloads/track_b/queries_c2_c5_v1.yaml` in the `yaseenahmed02/mcp-dbs-bench` repository, tagged at the pre-registration commit.

A.1 Track A: TPC-H reference queries

Track A uses five standard TPC-H queries from the TPC-H specification (Transaction Processing Performance Council, 2022) without modification. Query IDs match the TPC-H numbering. All queries are executed identically on Postgres, MySQL, and DuckDB via each MCP server's execute tool; the SQL text is the vendor-neutral form and each backend applies its own dialect adaptations at parse time.

Query	**Result cardinality**	**Structural stressor**
Q1	4 rows	1-table aggregation + `GROUP BY`
Q3	10 rows	3-way join + `ORDER BY` + `LIMIT`
Q6	1 row	1-table scan + scalar aggregate
Q9	∼175 rows	6-way join + 2-key `GROUP BY`
Q21	100 rows (`LIMIT 100`)	4-way join + correlated subquery

Every query holds its result cardinality fixed while T_{DB} grows with the scale factor, which isolates engine scaling from serialization scaling.

The executed SQL for all five queries follows, reproduced from the TPC-H Standard Specification rev. 3.0.1 (Transaction Processing Performance Council, 2022) in the PostgreSQL dialect; the MySQL and DuckDB variants in the repository differ only in date-arithmetic syntax. The specification is the normative source; any divergence in the executed SQL from Section 3.2.1 would be a bug in this thesis, not a redefinition of the query.

Q1: pricing summary report

```
SELECT
    l_returnflag,
    l_linestatus,
    SUM(l_quantity)                                       AS sum_qty,
    SUM(l_extendedprice)                                  AS sum_base_price,
    SUM(l_extendedprice * (1 - l_discount))               AS sum_disc_price,
    SUM(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
    AVG(l_quantity)                                       AS avg_qty,
    AVG(l_extendedprice)                                  AS avg_price,
    AVG(l_discount)                                       AS avg_disc,
    COUNT(*)                                              AS count_order
FROM lineitem
WHERE l_shipdate <= DATE '1998-12-01' - INTERVAL '90' DAY
GROUP BY l_returnflag, l_linestatus
ORDER BY l_returnflag, l_linestatus;
```

Q3: shipping priority

```
SELECT
    l_orderkey,
    SUM(l_extendedprice * (1 - l_discount)) AS revenue,
    o_orderdate,
    o_shippriority
FROM customer, orders, lineitem
WHERE c_mktsegment = 'BUILDING'
  AND c_custkey = o_custkey
  AND l_orderkey = o_orderkey
  AND o_orderdate < DATE '1995-03-15'
  AND l_shipdate > DATE '1995-03-15'
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY revenue DESC, o_orderdate
LIMIT 10;
```

Q6: forecasting revenue change

```
SELECT
    SUM(l_extendedprice * l_discount) AS revenue
FROM lineitem
WHERE l_shipdate >= DATE '1994-01-01'
  AND l_shipdate <  DATE '1994-01-01' + INTERVAL '1' YEAR
  AND l_discount BETWEEN 0.06 - 0.01 AND 0.06 + 0.01
  AND l_quantity < 24;
```

Q9: product type profit measure

```
SELECT
    nation,
    o_year,
    SUM(amount) AS sum_profit
FROM (
    SELECT
        n_name AS nation,
        EXTRACT(YEAR FROM o_orderdate)::INTEGER AS o_year,
        l_extendedprice * (1 - l_discount)
            - ps_supplycost * l_quantity AS amount
    FROM part, supplier, lineitem, partsupp, orders, nation
    WHERE s_suppkey = l_suppkey
      AND ps_suppkey = l_suppkey
      AND ps_partkey = l_partkey
      AND p_partkey = l_partkey
      AND o_orderkey = l_orderkey
      AND s_nationkey = n_nationkey
      AND p_name LIKE '%green%'
) AS profit
GROUP BY nation, o_year
ORDER BY nation, o_year DESC;
```

Q21: suppliers who kept orders waiting

```
SELECT
    s_name,
    COUNT(*) AS numwait
FROM supplier, lineitem AS l1, orders, nation
WHERE s_suppkey = l1.l_suppkey
```

```
  AND o_orderkey = l1.l_orderkey
  AND o_orderstatus = 'F'
  AND l1.l_receiptdate > l1.l_commitdate
  AND EXISTS (
      SELECT * FROM lineitem AS l2
      WHERE l2.l_orderkey = l1.l_orderkey
        AND l2.l_suppkey <> l1.l_suppkey
  )
  AND NOT EXISTS (
      SELECT * FROM lineitem AS l3
      WHERE l3.l_orderkey = l1.l_orderkey
        AND l3.l_suppkey <> l1.l_suppkey
        AND l3.l_receiptdate > l3.l_commitdate
  )
  AND s_nationkey = n_nationkey
  AND n_name = 'SAUDI ARABIA'
GROUP BY s_name
ORDER BY numwait DESC, s_name
LIMIT 100;
```

A.2 Track B: SQLyzr-C2 workload (15 queries)

Track B Category C2 queries exercise a single non-projection SQL construct beyond the basic `SELECT`: aggregation, single-predicate `WHERE`, a single equality join, `DISTINCT`, `ORDER BY`, or an unconditional `GROUP BY`. All fifteen queries are hand-written on the TPC-H schema and each was pre-registered under `workloads/track_b/queries_c2_c5_v1.yaml` before any LLM run.

Table A.1: Track B category C2 workload (15 hand-written natural-language queries pre-registered before any LLM run). `target_subcat` names the SQLyzr subcategory the query was designed to elicit.

Query ID	Subcat	Natural-language question
tb_c2_01	s6	How many customer records are in the database?
tb_c2_02	s6	What is the total value of all orders in the database (sum of o_totalprice)?
tb_c2_03	s6	What is the average retail price of parts?
tb_c2_04	s6	What is the minimum account balance among all suppliers?
tb_c2_05	s6	What is the maximum quantity value in any line item?
tb_c2_06	s7	Show 10 customer names from the customer table (any 10).

Table A.1 continued from previous page

Query ID	Subcat	Natural-language question
tb_c2_07	s8	List all distinct order statuses that appear in the orders table.
tb_c2_08	s8	List all distinct part manufacturers.
tb_c2_09	s3	List all nation names sorted alphabetically.
tb_c2_10	s4	Show the names of all customers whose account balance is negative.
tb_c2_11	s4	What is the name of the customer whose customer key is 100?
tb_c2_12	s4	List the names of suppliers based in nation with n_nationkey = 5.
tb_c2_13	s9	For each order priority, how many orders have that priority?
tb_c2_14	s9	For each market segment, how many customers belong to that segment?
tb_c2_15	s10	For each customer, show the customer name alongside their nation name.

A.3 Track B: SQLyzr-C5 workload (15 queries)

Track B Category C5 queries exercise nested subqueries combined with `GROUP BY` or aggregation, two-level nested subqueries, and `CASE` expressions. As in C2, all fifteen queries are hand-written on TPC-H and pre-registered.

Table A.2: Track B category C5 workload (15 hand-written natural-language queries pre-registered before any LLM run). `target_subcat` names the SQLyzr subcategory the query was designed to elicit.

Query ID	Subcat	Natural-language question
tb_c5_01	s28	For each nation (by n_nationkey), count how many customers have an account balance above the overall average customer balance.
tb_c5_02	s28	For each order priority, count orders whose total price exceeds the overall average order price.
tb_c5_03	s28	For each part manufacturer, count parts whose retail price is above the overall average retail price.
tb_c5_04	s29	List distinct customer names whose orders exceed the overall average order value.
tb_c5_05	s29	List distinct supplier names who supply at least one part whose supply cost is above the overall average supply cost.
tb_c5_06	s30	What is the total revenue (sum of o_totalprice) from orders placed by customers whose account balance exceeds the overall average customer balance?
tb_c5_07	s30	How many parts have retail price higher than the average retail price of parts made by manufacturer 'Manufacturer#1'?
tb_c5_08	s30	What is the average retail price of parts supplied by supplier with s_suppkey = 100?
tb_c5_09	s31	List distinct customer names whose orders have total price exceeding the overall average order total price.

Table A.2 continued from previous page

Query ID	Subcat	Natural-language question
tb_c5_10	s31	List distinct customer names who placed orders with total price greater than the average total price among orders placed by customers in nation 5.
tb_c5_11	s32	For each customer, show their name and label them 'high' if their account balance is greater than 5000, otherwise 'low'.
tb_c5_12	s32	For each part, show its name and label its price tier: 'premium' if retail price ¿ 2000, 'standard' if 500-2000, else 'budget'.
tb_c5_13	s32	Count how many customers are high-balance (c_acctbal ¿ 5000) and how many are low-balance (c_acctbal ¡= 5000), returning both counts in one row.
tb_c5_14	s32	For each nation (by n_nationkey), compute total account balance of customers, but only counting balances above 3000 (else count as 0).
tb_c5_15	s31	List distinct order priorities of orders placed by customers whose account balance is above the average balance of customers in nation 5.

A.4 Track B: gold SQL for all thirty questions

Each question ships with a pre-registered gold SQL query, frozen in `workloads/track_b/queries_c2_c5_v1.yaml` before any LLM call. The harness executes the gold query directly against the same database instance and stores the returned rows as ground truth (Section 4.5.1 gives the comparison rules). The full set follows, C2 then C5.

Category C2

tb_c2_01 (s6): *How many customer records are in the database?*

```
SELECT COUNT(*) FROM customer
```

tb_c2_02 (s6): *What is the total value of all orders in the database (sum of o_totalprice)?*

```
SELECT SUM(o_totalprice) FROM orders
```

tb_c2_03 (s6): *What is the average retail price of parts?*

```
SELECT AVG(p_retailprice) FROM part
```

tb_c2_04 (s6): *What is the minimum account balance among all suppliers?*

```
SELECT MIN(s_acctbal) FROM supplier
```

tb_c2_05 (s6): *What is the maximum quantity value in any line item?*

```
SELECT MAX(l_quantity) FROM lineitem
```

tb_c2_06 (s7): *Show 10 customer names from the customer table (any 10).*

```
SELECT c_name FROM customer LIMIT 10
```

tb_c2_07 (s8): *List all distinct order statuses that appear in the orders table.*

```
SELECT DISTINCT o_orderstatus FROM orders
```

tb_c2_08 (s8): *List all distinct part manufacturers.*

```
SELECT DISTINCT p_mfgr FROM part
```

tb_c2_09 (s3): *List all nation names sorted alphabetically.*

```
SELECT n_name FROM nation ORDER BY n_name
```

tb_c2_10 (s4): *Show the names of all customers whose account balance is negative.*

```
SELECT c_name FROM customer WHERE c_acctbal < 0
```

tb_c2_11 (s4): *What is the name of the customer whose customer key is 100?*

```
SELECT c_name FROM customer WHERE c_custkey = 100
```

tb_c2_12 (s4): *List the names of suppliers based in nation with n_nationkey = 5.*

```
SELECT s_name FROM supplier WHERE s_nationkey = 5
```

tb_c2_13 (s9): *For each order priority, how many orders have that priority?*

```
SELECT o_orderpriority, COUNT(*) FROM orders GROUP BY o_orderpriority
```

tb_c2_14 (s9): *For each market segment, how many customers belong to that segment?*

```
SELECT c_mktsegment, COUNT(*) FROM customer GROUP BY c_mktsegment
```

tb_c2_15 (s10): *For each customer, show the customer name alongside their nation name.*

```
SELECT c_name, n_name FROM customer, nation WHERE c_nationkey = n_nationkey
```

Category C5

tb_c5_01 (s28): *For each nation (by n_nationkey), count how many customers have an account balance above the overall average customer balance.*

```
SELECT c_nationkey, COUNT(*)
FROM customer
WHERE c_acctbal > (SELECT AVG(c_acctbal) FROM customer)
GROUP BY c_nationkey
```

tb_c5_02 (s28): *For each order priority, count orders whose total price exceeds the overall average order price.*

```
SELECT o_orderpriority, COUNT(*)
FROM orders
WHERE o_totalprice > (SELECT AVG(o_totalprice) FROM orders)
GROUP BY o_orderpriority
```

tb_c5_03 (s28): *For each part manufacturer, count parts whose retail price is above the overall average retail price.*

```
SELECT p_mfgr, COUNT(*)
FROM part
WHERE p_retailprice > (SELECT AVG(p_retailprice) FROM part)
GROUP BY p_mfgr
```

tb_c5_04 (s29): *List distinct customer names whose orders exceed the overall average order value.*

```
SELECT DISTINCT c_name
FROM customer, orders
WHERE c_custkey = o_custkey
  AND o_totalprice > (SELECT AVG(o_totalprice) FROM orders)
```

tb_c5_05 (s29): *List distinct supplier names who supply at least one part whose supply cost is above the overall average supply cost.*

```
SELECT DISTINCT s_name
FROM supplier, partsupp
WHERE s_suppkey = ps_suppkey
  AND ps_supplycost > (SELECT AVG(ps_supplycost) FROM partsupp)
```

tb_c5_06 (s30): *What is the total revenue (sum of o_totalprice) from orders placed by customers whose account balance exceeds the overall average customer balance?*

```
SELECT SUM(o_totalprice)
FROM orders
WHERE o_custkey IN (SELECT c_custkey FROM customer WHERE c_acctbal > (SELECT AVG(
   c_acctbal) FROM customer))
```

tb_c5_07 (s30): *How many parts have retail price higher than the average retail price of parts made by manufacturer 'Manufacturer#1'?*

```
SELECT COUNT(*)
FROM part
WHERE p_retailprice > (SELECT AVG(p_retailprice) FROM part WHERE p_mfgr = 'Manufacturer
   #1')
```

tb_c5_08 (s30): *What is the average retail price of parts supplied by supplier with s_suppkey = 100?*

```
SELECT AVG(p_retailprice)
FROM part
WHERE p_partkey IN (SELECT ps_partkey FROM partsupp WHERE ps_suppkey = 100)
```

tb_c5_09 (s31): *List distinct customer names whose orders have total price exceeding the overall average order total price.*

```
SELECT DISTINCT c_name
FROM customer
WHERE c_custkey IN (
  SELECT o_custkey FROM orders WHERE o_totalprice > (
    SELECT AVG(o_totalprice) FROM orders
  )
)
```

tb_c5_10 (s31): *List distinct customer names who placed orders with total price greater than the average total price among orders placed by customers in nation 5.*

```
SELECT DISTINCT c_name
FROM customer, orders
WHERE c_custkey = o_custkey
  AND o_totalprice > (
    SELECT AVG(o_totalprice) FROM orders WHERE o_custkey IN (
      SELECT c_custkey FROM customer WHERE c_nationkey = 5
    )
  )
```

tb_c5_11 (s32): *For each customer, show their name and label them 'high' if their account balance is greater than 5000, otherwise 'low'.*

```
SELECT c_name, CASE WHEN c_acctbal > 5000 THEN 'high' ELSE 'low' END AS tier FROM
   customer
```

tb_c5_12 (s32): *For each part, show its name and label its price tier: 'premium' if retail price ¿ 2000, 'standard' if 500-2000, else 'budget'.*

```
SELECT p_name,
  CASE
    WHEN p_retailprice > 2000 THEN 'premium'
    WHEN p_retailprice >= 500 THEN 'standard'
```

```
    ELSE 'budget'
  END AS tier
FROM part
```

tb_c5_13 (s32): *Count how many customers are high-balance (c_acctbal ¿ 5000) and how many are low-balance (c_acctbal ¡= 5000), returning both counts in one row.*

```
SELECT
  SUM(CASE WHEN c_acctbal > 5000 THEN 1 ELSE 0 END) AS high_balance,
  SUM(CASE WHEN c_acctbal <= 5000 THEN 1 ELSE 0 END) AS low_balance
FROM customer
```

tb_c5_14 (s32): *For each nation (by n_nationkey), compute total account balance of customers, but only counting balances above 3000 (else count as 0).*

```
SELECT c_nationkey,
  SUM(CASE WHEN c_acctbal > 3000 THEN c_acctbal ELSE 0 END) AS eligible_balance
FROM customer
GROUP BY c_nationkey
```

tb_c5_15 (s31): *List distinct order priorities of orders placed by customers whose account balance is above the average balance of customers in nation 5.*

```
SELECT DISTINCT o_orderpriority
FROM orders
WHERE o_custkey IN (
  SELECT c_custkey FROM customer WHERE c_acctbal > (
    SELECT AVG(c_acctbal) FROM customer WHERE c_nationkey = 5
  )
)
```

A.5 Track B: agent system prompt

The following prompt is fixed across all servers and categories. It is loaded by content hash into every Track B run manifest; changing it requires a benchmark version bump.

```
You are an SQL agent answering natural-language questions about the beaver
data warehouse via an MCP-mediated database server. Your goal is to produce
the answer to the user's question, returned strictly as a JSON object using
the submit_answer tool.

Operational rules:

  1. The MCP server exposes a small set of tools you can call: tools/list will
```

```
   have been called for you already; the discovered tool manifest is
   summarized in the user message.
2. You may issue at most 10 tool calls in total. If you do not call
   submit_answer within 10 tool calls, the task is marked
   call-limit-exceeded.
3. When a tool call fails (SQL syntax error, schema grounding error,
   timeout), you may retry the same logical query with a corrected version,
   up to 3 retries per logical query.
4. Do not hallucinate table or column names. If you are uncertain about the
   schema, call a discovery or describe tool first.
5. Produce the final answer by calling submit_answer with a JSON-serializable
   "result" value matching the expected result shape for the question.
   The result must be the rows (list of lists) the SQL would return, with
   numeric tolerance handled by the harness.
6. Be terse. Do not produce explanatory prose; produce tool calls and a
   final submit_answer.

Version: 1 (Wave 2b / methodology Addendum A4). Changing this prompt requires a
benchmark version bump; the file is loaded by hash into every Track B run
manifest.
```

Glossary

Protocol Tax The total overhead introduced by the MCP middleware layer, defined as the difference between end-to-end agentic query latency and raw database execution time: $\Delta P = T_{\text{total}} - T_{\text{DB_exec}}$

Knowledge Shipping The paradigm in which a database system transmits not only raw result rows but also schema metadata, constraints, type information, and relationship descriptions to an LLM agent in a text-based format suitable for reasoning

Serialization Choke Point The data volume threshold at which JSON-RPC serialization time in the MCP server becomes the dominant component of end-to-end latency, causing the database engine to sit idle while the protocol layer processes results

Implementation Metric A performance metric whose variance across MCP server implementations is attributable to specific developer design decisions (e.g., schema verbosity, tool granularity, caching strategy) rather than to the database engine or runtime environment

Coupling Metric A performance metric that measures the integrated behaviour of the full DBMS–MCP stack, where observed variance cannot be attributed to a single component without further decomposition