THE POWER OF TWO: HOW RAG ELEVATES TEXT-TO-SQL ACCURACY

A Practical Approach for Enterprise-Scale Databases

Tech Talk Presentation | February 2026

THE TRADITIONAL APPROACH FALLS SHORT

- ❌ Injecting entire schema into every prompt
- ❌ Context window overflow (50+ tables × 20+ columns)
- ❌ Token waste & reduced accuracy

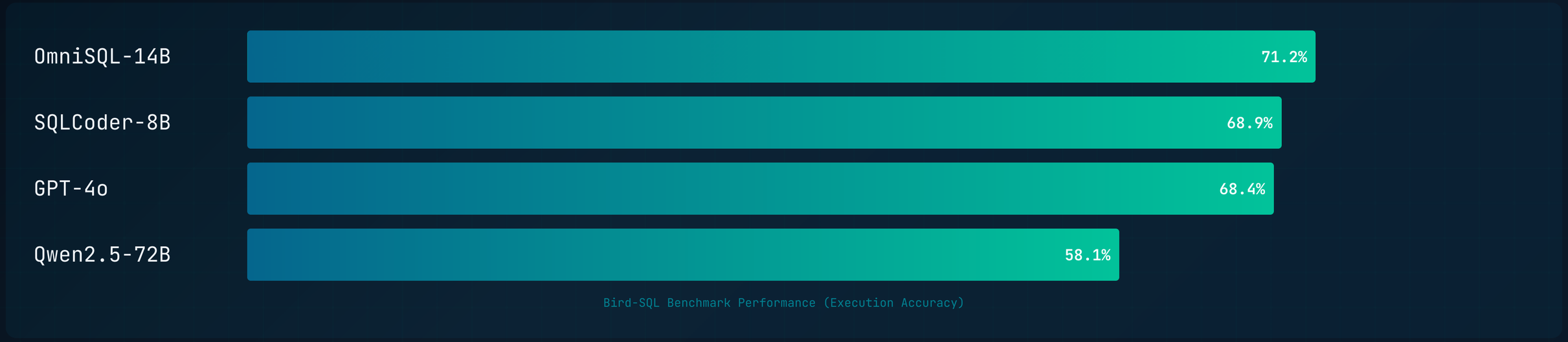
SCHEMA LINKING: RETRIEVE, DON'T INJECT ALL

- ✅ Enrich schemas with semantic descriptions
- ✅ Index into Vector Database
- ✅ Retrieve only top-K relevant tables per query

1 Enrich Schema Files → **2** Index Vector DB → **3** Retrieve Relevant Tables

NOT ALL LLMS ARE CREATED EQUAL FOR SQL

- General-purpose models (Llama, Qwen, GPT) excel at chat
- SQL generation requires specialized reasoning
- Benchmark evidence matters



ENTER SQL-SPECIALIZED LLMS

71.2%
OMNISQL BIRD-SQL

87.1%
OMNISQL SPIDER

14B
PARAMETERS

✓
SELF-HOSTABLE

MODEL	SIZE	BIRD-SQL EX	SPIDER EX	SELF-HOST
OmniSQL	14B	71.2%	87.1%	✅ Ollama
SQLCoder	8B	68.9%	84.5%	✅ Ollama
GPT-4o	?	68.4%	85.3%	❌ API
Qwen2.5	72B	58.1%	79.2%	✅ Heavy

OMNISQL BENCHMARK

Table 5: Main results on 9 datasets (%). The best results are highlighted in bold. "DSC" is the abbreviation of "DeepSeek-Coder".

LLM	Spider (dev)		Spider (test)		BIRD (dev)		Spider2.0-SQLite		Science Benchmark		EHRSQL		Spider-DK		Spider-Syn		Spider-Realistic		Average	
	Gre	Maj	Gre	Maj	Gre	Maj	Gre	Maj	Gre	Maj	Gre	Maj	Gre	Maj	Gre	Maj	Gre	Maj	Gre	Maj
Closed-source LLMs (as a reference)																				
GPT-4o-mini	70.4	71.0	82.4	83.7	58.8	61.5	5.9	7.4	51.8	52.5	37.9	43.1	73.3	74.4	60.5	61.6	64.4	66.7	56.2	58.0
GPT-4-Turbo	72.4	72.2	83.4	84.2	62.0	63.6	11.9	11.9	59.2	59.5	43.1	44.8	72.3	72.1	62.9	63.5	67.5	68.3	59.4	60.0
GPT-4o	70.9	70.7	83.2	84.9	61.9	64.0	14.8	15.6	55.5	56.2	44.9	45.5	72.9	73.5	59.6	62.3	66.5	66.7	58.9	59.9
Open-source LLMs (~7B)																				
DSC-6.7B-Instruct	63.2	63.2	70.5	73.2	43.1	48.0	3.0	3.7	40.8	45.5	28.6	33.9	60.9	64.1	49.9	51.7	58.7	58.9	46.5	49.1
Qwen2.5-Coder-7B-Instruct	73.4	77.1	82.2	85.6	50.9	61.3	1.5	2.2	45.2	51.2	24.3	36.9	67.5	73.6	63.1	66.9	66.7	70.5	52.8	58.4
Qwen2.5-7B-Instruct	65.4	68.9	76.8	82.6	46.9	56.4	1.5	1.5	38.5	47.5	20.9	32.1	63.7	71.8	54.2	60.0	56.7	63.6	47.2	53.8
OpenCoder-8B-Instruct	59.5	59.5	68.3	70.1	37.5	45.3	0.7	0.7	39.8	45.5	21.9	29.9	62.6	64.7	46.0	46.1	49.0	49.4	42.8	45.7
Meta-Llama-3.1-8B-Instruct	61.8	67.7	72.2	78.5	42.0	53.1	3.0	2.2	36.8	43.1	24.6	33.7	62.6	69.9	53.1	59.3	57.5	61.0	46.0	52.1
Granite-8B-Code-Instruct	58.5	59.2	64.9	68.6	27.6	32.5	0.7	0.7	29.4	31.4	16.0	22.6	50.7	54.4	45.0	46.8	48.8	49.4	38.0	40.6
Granite-3.1-8B-Instruct	58.3	65.0	69.8	75.3	36.0	47.2	1.5	1.5	36.8	47.5	19.6	32.3	60.0	66.5	47.7	53.8	46.5	57.1	41.8	49.6
OmniSQL-7B (Ours)	**81.2**	**81.6**	**87.9**	**88.9**	**63.9**	**66.1**	**10.4**	**10.4**	**50.2**	**55.9**	**34.9**	**40.0**	**76.1**	**77.8**	**69.7**	**69.6**	**76.2**	**78.0**	**61.2**	**63.1**
Open-source LLMs (14B-32B)																				
Qwen2.5-Coder-14B-Instruct	78.1	80.6	86.6	88.0	61.5	**66.1**	5.9	4.4	52.2	54.2	31.6	35.5	**73.6**	**77.8**	68.2	69.3	76.2	74.2	59.3	61.1
Qwen2.5-14B-Instruct	66.5	69.7	82.0	84.0	56.7	62.1	5.2	10.4	51.2	56.2	28.8	35.2	72.3	74.0	58.1	60.7	62.4	65.2	53.7	57.5
Starcoder2-15B-Instruct	65.8	67.6	73.0	74.0	38.5	42.6	0.7	3.0	25.8	29.8	16.8	22.6	66.5	68.2	49.4	52.4	56.7	61.0	43.7	46.8
DSC-V2-Lite-Inst. (16B, MoE)	68.0	70.0	77.9	79.6	44.6	51.8	3.0	5.2	39.1	45.8	23.9	32.4	63.7	67.5	55.6	57.9	61.8	64.0	48.6	52.7
Granite-20B-Code-Instruct	65.7	63.6	74.1	72.9	34.0	40.5	0.0	0.0	37.5	40.1	23.5	26.9	62.2	63.9	52.3	54.3	55.7	56.3	45.0	46.5
Codestral-22B	66.7	67.5	78.6	81.0	52.7	56.8	5.2	5.9	48.5	54.2	37.8	40.4	69.9	72.7	55.2	59.4	62.6	64.8	53.0	55.9
OmniSQL-14B (Ours)	**81.4**	**82.0**	**88.3**	**88.3**	**64.2**	65.9	**10.4**	**13.3**	**56.9**	**56.9**	**39.9**	**43.6**	72.9	74.8	**69.0**	**72.0**	**76.4**	**78.5**	**62.2**	**63.9**
Open-source LLMs (≥ 32B)																				
Qwen2.5-Coder-32B-Instruct	77.7	77.9	87.5	88.0	64.5	67.0	5.9	11.9	54.8	56.5	36.4	43.3	**78.3**	**78.1**	**69.9**	70.5	72.4	74.8	60.8	63.1
Qwen2.5-32B-Instruct	71.9	73.6	84.9	86.1	62.0	64.7	7.4	8.9	50.5	54.5	33.6	41.4	73.1	76.1	64.0	66.0	66.5	68.1	57.1	59.9
DSC-33B-Instruct	66.0	68.5	74.3	76.5	49.2	55.9	7.4	4.4	44.5	52.2	31.4	35.4	69.0	71.4	53.5	57.4	59.1	63.2	50.5	53.9
Granite-34B-Code-Instruct	69.9	70.0	74.4	77.0	33.8	41.3	0.7	1.5	40.1	40.1	23.8	29.9	64.7	70.7	55.6	59.8	60.0	59.6	47.0	50.0
Mixtral-8x7B-Inst. (47B, MoE)	54.4	59.0	67.8	74.1	35.3	42.9	2.2	2.2	29.4	34.8	21.5	31.4	55.3	60.4	42.1	48.8	48.0	53.3	39.6	45.2
Meta-Llama-3.1-70B-Instruct	72.3	71.0	84.3	85.9	**65.1**	**67.4**	3.0	3.7	55.2	56.2	37.4	41.4	75.1	**78.1**	61.7	63.1	64.0	65.6	57.6	59.2
Qwen2.5-72B-Instruct	73.9	72.1	84.0	85.7	60.3	63.6	9.6	14.8	52.8	58.2	35.0	41.2	76.4	77.6	64.1	64.3	70.1	68.5	58.5	60.7
DeepSeek-V3 (671B, MoE)	73.1	73.5	85.5	85.8	63.2	63.8	**12.6**	**15.6**	56.2	57.9	**43.2**	43.5	72.9	73.8	64.4	65.1	67.9	66.9	59.9	60.7
OmniSQL-32B (Ours)	**80.9**	**80.9**	**87.6**	**89.8**	64.5	67.0	11.9	13.3	**57.2**	**58.5**	42.4	**46.8**	76.1	77.6	69.7	**72.1**	**78.1**	**77.2**	**63.2**	**64.8**

OmniSQL outperforms general-purpose LLMs by 3-13% on SQL benchmarks

END-TO-END SYSTEM ARCHITECTURE

1. User Query Input
2. Query Enhancement
3. Column-First RAG
4. Anchor Tables
5. Table Expansion
6. Schema Context
7. SQL Generation
8. T-SQL Correction
9. Self-Healing
10. Response Format

Complete pipeline from natural language to validated T-SQL with automated error recovery

AUTO-ENRICHMENT WITH HUMAN REVIEW

BEFORE (Raw DDL)

```
CREATE TABLE Person (
PersonID INT PRIMARY KEY,
FirstName NVARCHAR(50),
Gender NVARCHAR(1)
);
```

AFTER (Enriched)

```
Table: Person
Description: "Core entity storing
employee and contact information"

Columns:
- PersonID: Unique identifier (PK)
- FirstName: Given name
- Gender: M=Male, F=Female, U=Unknown
Tags: ["sex", "sexual"]
```

DUAL-COLLECTION INDEXING STRATEGY

Collection 1: Column Metadata

Fine-grained column-level indexing

`Embed: "column_name + description + tags"`

Collection 2: Table Metadata

Coarse-grained table-level indexing

`Embed: "table_name + semantic_description"`

Separate embedding spaces enable both precision and recall optimization

WHY COLUMN-FIRST SEARCH?

- User queries often mention column-level concepts
- More granular matching = higher recall
- Columns aggregate to tables (bottom-up approach)

Column-First ✓

```
Query: "orders in January 2024"

1. Search columns →
"OrderDate" (0.89)
"PlacedDate" (0.85)

2. Aggregate to tables →
SalesOrder (2 hits)

3. Return: SalesOrder
```

Table-First ✗

```
Query: "orders in January 2024"

1. Search tables →
"SalesOrderHeader" (0.45)
(missed due to name mismatch)

❌ May miss relevant table
because "orders" doesn't
match "SalesOrderHeader"
```

FROM COLUMNS TO ANCHOR TABLES

Step 1: For each column hit, aggregate score to parent table

↓

Step 2: Sort tables by aggregated score (descending)

↓

Step 3: Return top-K tables as Anchors (default K=5)

TABLE	COLUMN HITS	AGGREGATE SCORE
SalesOrderHeader	OrderDate (0.89) + Status (0.72)	**1.61**
Customer	Name (0.65)	0.65
Product	(none)	0.00

FINDING RELATED TABLES VIA FK TRAVERSAL

Scoring Formula

```
Related Score = Base Score + Bridge Bonus + Keyword Bonus

Base Score = 0.5 × max(AnchorScore)
Bridge Bonus = +1.0 if connected to >1 anchor
Keyword Bonus = +0.5 if matches query tokens
```

SMART CONTEXT OPTIMIZATION

BEFORE (Full Schema)

```
CREATE TABLE SalesOrder (
OrderID INT PRIMARY KEY,
RowGuid uniqueidentifier,
ModifiedDate datetime,
CreatedBy int,
OrderDate DATE,
Status VARCHAR(20)
);
```

AFTER (Optimized)

```
CREATE TABLE SalesOrder (
OrderID INT PRIMARY KEY,
OrderDate DATE,
Status VARCHAR(20)
-- Samples: ['Pending',
'Shipped',
'Cancelled']
);
```

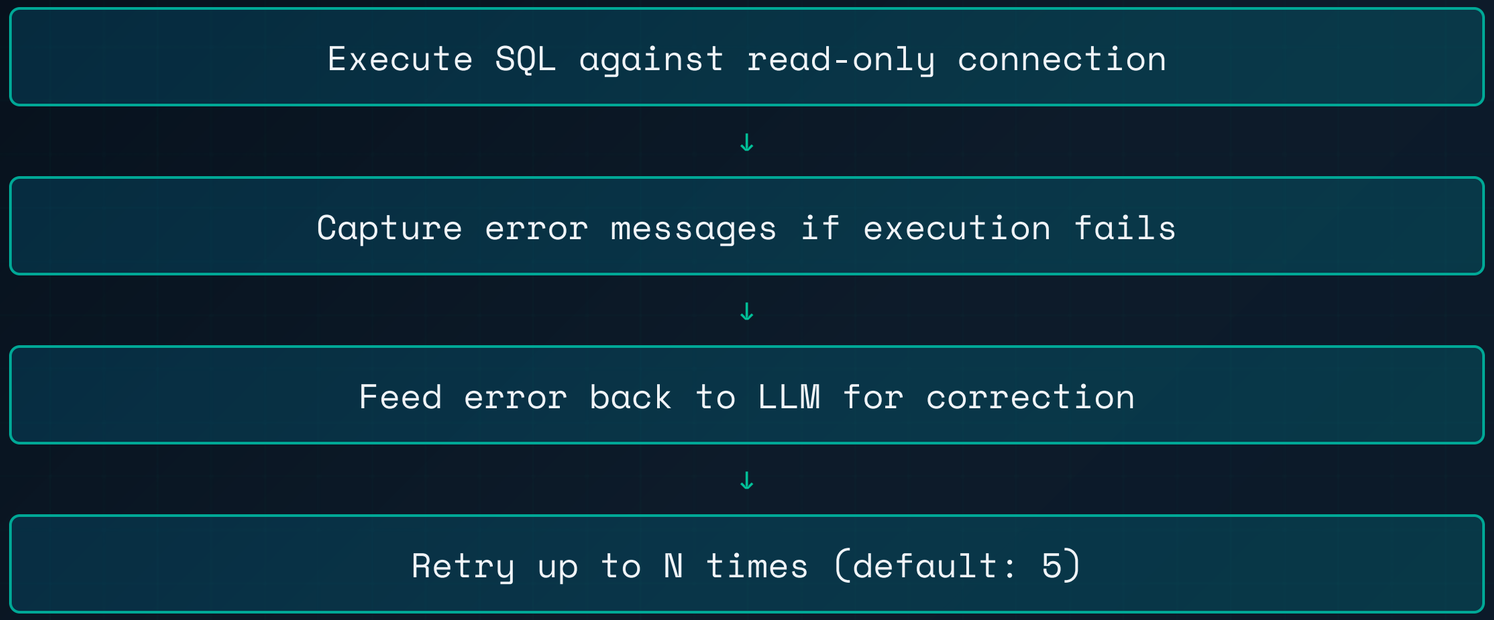
- ✅ Pruning: Remove low-value columns (IDs, timestamps, system cols)
- ✅ Sample Injection: Add real values for search columns

ENFORCING T-SQL SYNTAX COMPLIANCE

MYSQL/POSTGRESQL	T-SQL CORRECTION
LIMIT 10	SELECT TOP 10
string1 || string2	CONCAT(string1, string2)
NOW()	GETDATE()
IF(cond, a, b)	IIF(cond, a, b)
table.group	table.[group]

Rule-based validator + transformer auto-corrects common mistakes before execution

EXECUTE, DIAGNOSE, HEAL, RETRY




```
Attempt 1: "Invalid object name 'Orders'"
→ LLM Fix: Add schema prefix "Sales.Orders"

Attempt 2: "Invalid column name 'Qty'"
```

⚠️ **Security Note: Always use READ-ONLY database credentials**

BUILDING YOUR EVALUATION GOLD STANDARD

COLUMN	PURPOSE
question	Natural language query
gold_sql	Human-verified correct SQL
gold_tables	Tables that should be retrieved
expected_record_count	Number of rows expected
expected_pks	Primary key values for verification
difficulty	basic / intermediate / advanced

Generate question-SQL pairs covering all difficulty levels with expected results & metadata. Human review is mandatory.

BASIC SQL DATASET SAMPLES

#	QUESTION	QUERY	TABLES
1	List all departments	`SELECT * FROM HumanResources.Department;`	HumanResources.Department
6	Retrieve names of all products	`SELECT Name FROM Production.Product;`	Production.Product
11	Get total employees in company	`SELECT COUNT(*) FROM HumanResources.Employee;`	HumanResources.Employee
18	Find highest list price	`SELECT MAX(ListPrice) FROM Production.Product;`	Production.Product
23	Find all departments in manufacturing	`SELECT * FROM HumanResources.Department WHERE GroupName = 'Manufacturing';`	HumanResources.Department

✅ **Basic SQL Queries | Single-table operations | Filtering & Aggregation**

INTERMEDIATE SQL DATASET SAMPLES

#	QUESTION	QUERY	TABLES
201	Total sales per territory	SELECT st.Name, SUM(soh.SubTotal) AS TotalSales FROM Sales.SalesOrderHeader soh JOIN Sales.SalesTerritory st ON soh.TerritoryID = st.TerritoryID GROUP BY st.Name;	SalesOrderHeader, SalesTerritory
207	Calculate running total of sales	SELECT SalesOrderID, OrderDate, SubTotal, SUM(SubTotal) OVER(ORDER BY OrderDate, SalesOrderID) AS RunningTotal FROM Sales.SalesOrderHeader;	SalesOrderHeader
215	Products sold in >10 orders	SELECT ProductID, COUNT(DISTINCT SalesOrderID) AS OrderCount FROM Sales.SalesOrderDetail GROUP BY ProductID HAVING COUNT(DISTINCT SalesOrderID) > 10;	SalesOrderDetail
	Calculate total	WITH YearlySales AS (SELECT YEAR(OrderDate) AS SaleYear, TotalDue FROM	

✅ **Intermediate Queries | JOINs | Window Functions | CTEs | Subqueries**

ADVANCED SQL DATASET SAMPLES

#	QUESTION	QUERY	TABLES
301	Management hierarchy (Recursive CTE)	WITH Emp_CTE AS (SELECT BusinessEntityID, OrganizationNode, OrganizationLevel, JobTitle, CAST(JobTitle AS nvarchar(MAX)) AS HierarchyPath FROM HumanResources.Employee WHERE OrganizationLevel = 0 UNION ALL SELECT e.BusinessEntityID, e.OrganizationNode, e.OrganizationLevel, e.JobTitle, ecte.HierarchyPath + ' → ' + e.JobTitle FROM HumanResources.Employee e INNER JOIN Emp_CTE ecte ON e.OrganizationNode.GetAncestor(1) = ecte.OrganizationNode) SELECT * FROM Emp_CTE;	HumanResources.Employee
303	Pivot sales by year and territory	SELECT * FROM (SELECT YEAR(soh.OrderDate) AS [Year], st.[Group], soh.TotalDue FROM Sales.SalesOrderHeader soh JOIN Sales.SalesTerritory st ON soh.TerritoryID = st.TerritoryID) AS SourceTable PIVOT (SUM(TotalDue) FOR [Group] IN ([North America], [Europe], [Pacific])) AS PivotTable;	SalesOrderHeader, SalesTerritory
	3-month	WITH MonthlySales AS (SELECT CAST(FORMAT(OrderDate, 'yyyy-MM-01') AS date) AS	

✅ **Advanced Queries | Recursive CTEs | PIVOT/UNPIVOT | XML | Analytics**

CONTINUOUS LEARNING FROM EXPERT FEEDBACK

User asks question → System generates SQL

↓

Expert reviews → Clicks 👍 or 👎

↓

If 👍: Index (question, SQL) pair into gold dataset vector store

↓

Future queries → Retrieve similar pairs as few-shot examples

↓

LLM generates SQL with reference examples

KEY METRICS FOR SYSTEM EVALUATION

Table Recall

Percentage of gold tables retrieved by RAG

Table Recall = |Retrieved ∩ Gold| / |Gold|

Execution Success Rate

Percentage of queries that execute without error

Execution Success = Successful Queries / Total Queries

Semantic Score

AST-based SQL similarity measurement

Semantic Score = 1 - (AST Diff Nodes / Total AST Nodes)

ADDITIONAL EVALUATION METRICS

Record Count Match

Verifies the generated SQL returns the correct number of rows

Match = (Generated Row Count == Expected Row Count)

> Example:
> Expected: 150 orders | Generated SQL: 150 rows → ✅ Match
> Expected: 150 orders | Generated SQL: 148 rows → ❌ Mismatch

Expected Answer Match

Compares actual result values with ground truth answers

Match = (Generated Results ⊆ Expected Results)

> Example:
> Expected: Total Sales = $1,234,567.89
> Generated: Total Sales = $1,234,567.89 → ✅ Match
> Generated: Total Sales = $1,234,500.00 → ❌ Mismatch

SYSTEM PERFORMANCE RESULTS

377	96.16%	88.86%	66.44%
TOTAL QUERIES	TABLE RECALL	EXECUTION SUCCESS	SEMANTIC SCORE

DIFFICULTY	COUNT	TABLE RECALL	EXEC SUCCESS	SEMANTIC SCORE
Basic	192	99.39%	98.44%	73.60%
Intermediate	97	93.28%	87.63%	62.65%
Advanced	88	92.27%	69.32%	54.98%

PROVING THE VALUE OF RAG

Key Insight: High table recall (96%+) enables accurate SQL generation

Each 5% improvement in recall ≈ 8% improvement in execution success

PERFORMANCE & LATENCY ANALYSIS

STAGE	AVG (MS)	MEDIAN (MS)	MAX (MS)
Retrieval	130	101	845
Generation	35,352	35,800	88,464
Execution	68	39	1,676
Total	35,556	35,919	88,707

⚠️ Key Observation: Generation dominates latency (self-hosted LLM)
Retrieval + Execution: <200ms | Generation: ~35 seconds

UNDERSTANDING FAILURE CASES

19
RUNTIME ERRORS (45%)

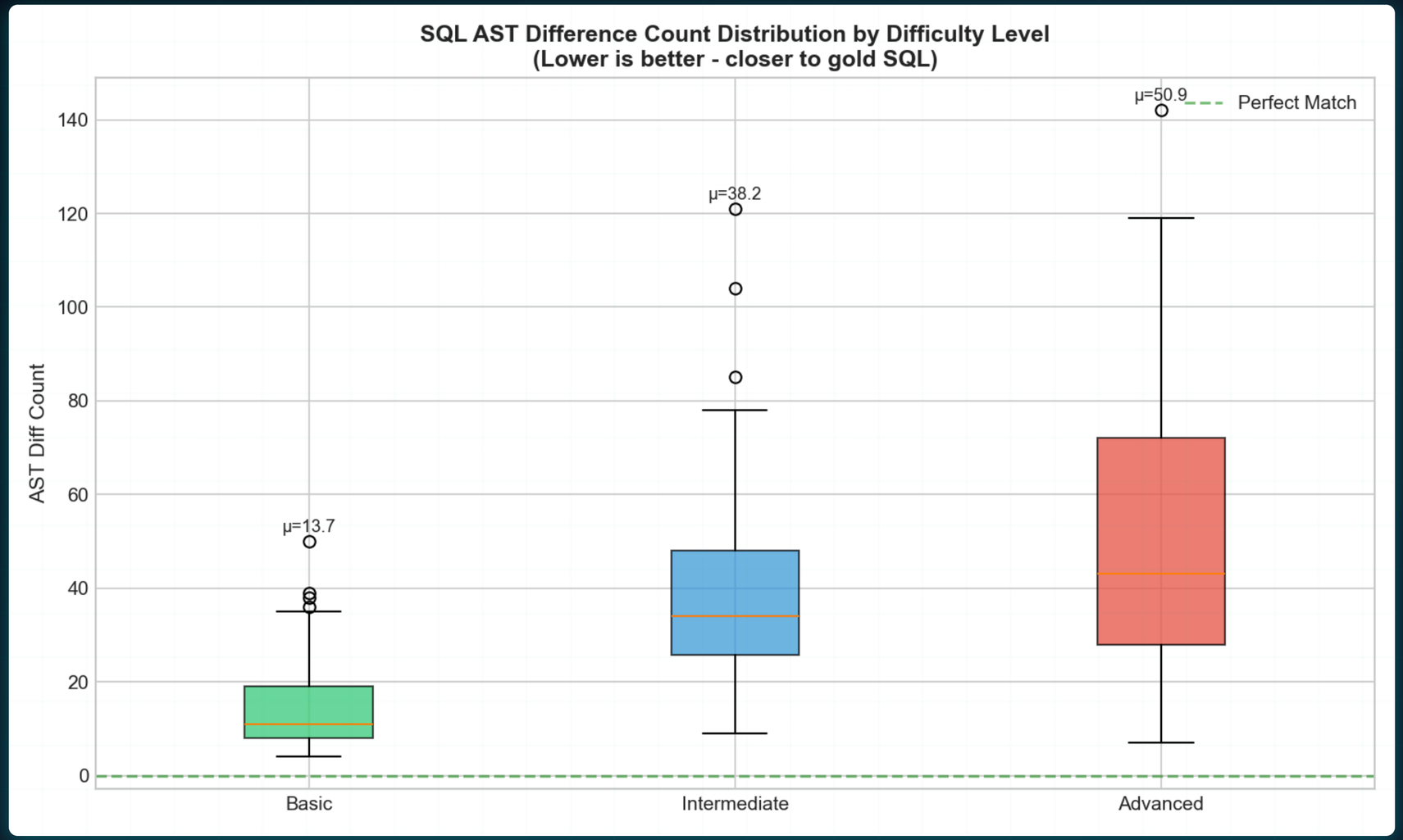
15
DRIVER ERRORS (36%)

8
SYNTAX ERRORS (19%)

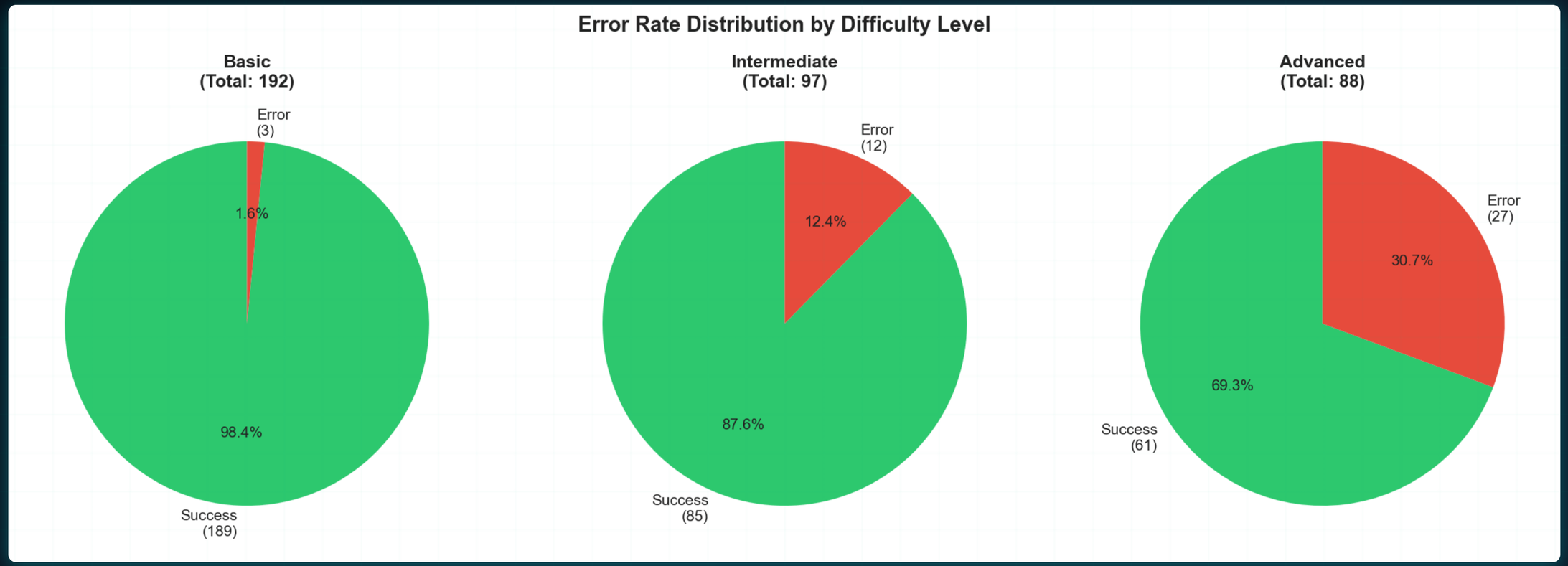
DIFFICULTY LEVEL	DRIVER	RUNTIME	SYNTAX
Basic	1	2	0
Intermediate	3	8	1
Advanced	11	9	7

Advanced queries account for 64% of all errors despite being only 23% of queries

SQL AST DIFFERENCE COUNT DISTRIBUTION



ERROR RATE DISTRIBUTION



RETRY DISTRIBUTION

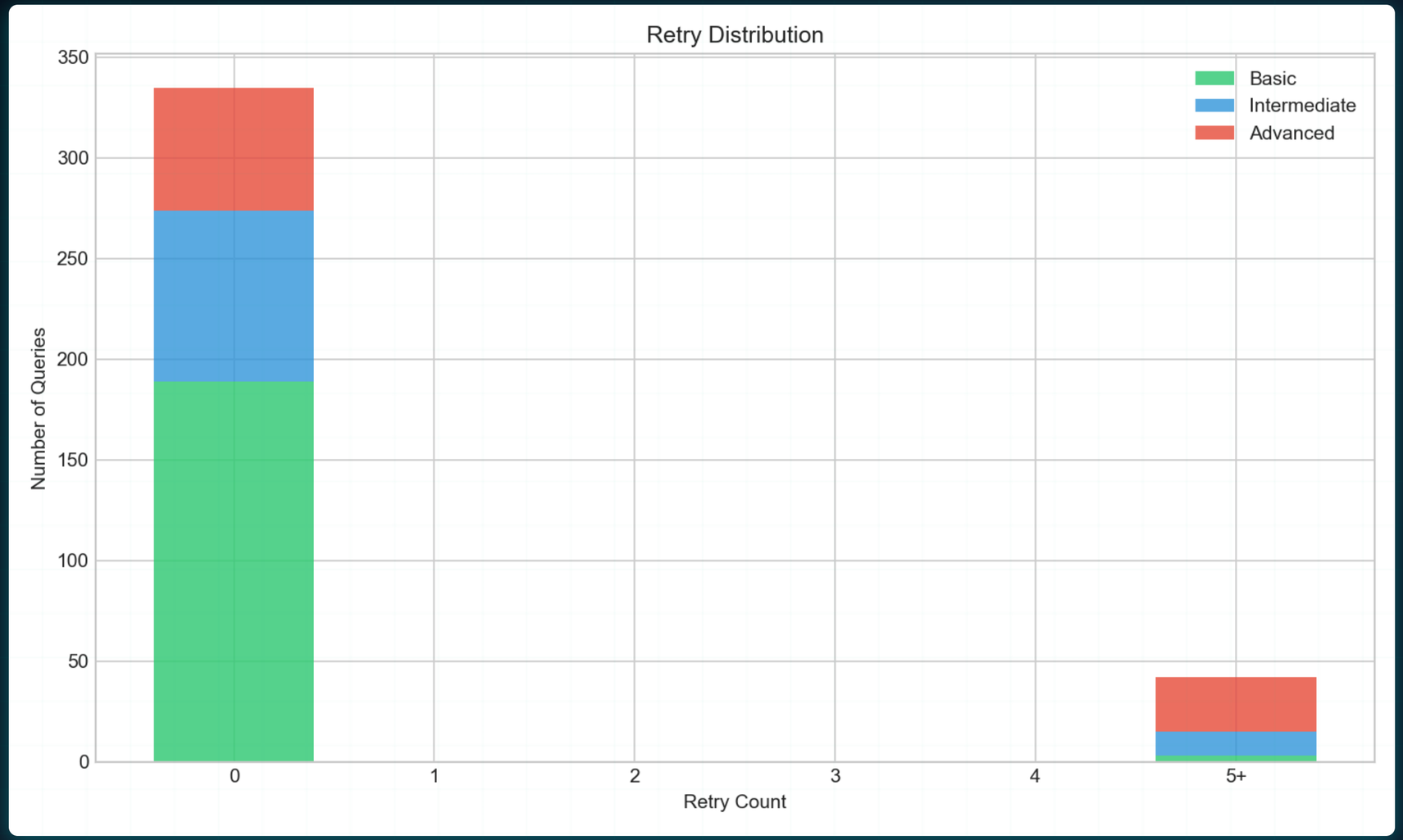
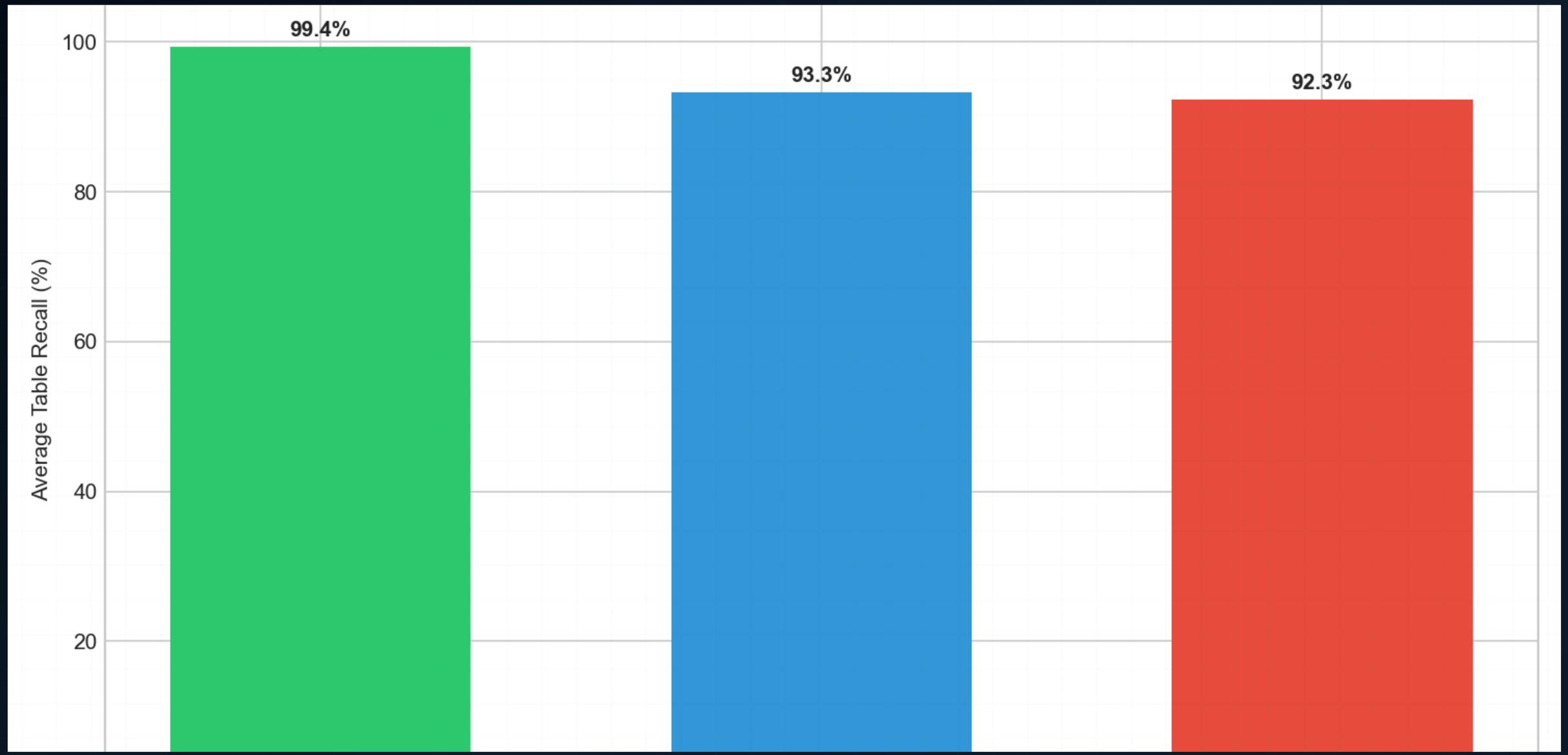


TABLE RECALL AVERAGE



ROADMAP FOR ENHANCED ACCURACY

- **Multi-hop Reasoning:** 2-hop FK traversal for complex JOINs
- **Reranker Integration:** Cross-encoder reranking for table selection
- **Query Decomposition:** Break complex questions into sub-queries
- **Dialect Fine-tuning:** Custom T-SQL fine-tuned model
- **Caching Layer:** LRU cache for common query patterns

> **Target: 95%+ execution success rate across all difficulty levels**

KEY TAKEAWAYS

- ✅ RAG-based Schema Linking solves context overflow
- ✅ SQL-specialized LLMs outperform general models
- ✅ Column-first retrieval improves recall accuracy
- ✅ Self-healing loop handles edge cases
- ✅ Continuous feedback enables improvement

96%	89%	377
TABLE RECALL	EXECUTION SUCCESS	QUERIES TESTED

Questions?