RUBICON: Agentic AI for Messy Enterprise Data

Fabian Wenz[1,2] Felix Treutwein[4] Çağatay Demiralp[3,2] Michael Stonebraker[2]
[1]TU-Darmstadt [2]MIT [3]AWS AI Labs [4]Landeshauptstadt München
fabian.wenz@tu-darmstadt.de,felix.treutwein@muenchen.de,{cagatay,stonebraker}@csail.mit.edu

ABSTRACT

Enterprise data exists in many forms, such as tables, text, maps, e-mail, and CAD models, that are invariably access-controlled and behind bespoke interfaces. Current agentic AI systems delegate the entire query workflow to a frontier LLM: a single model is left to interpret the request, decide which sources or tools to use, integrate the evidence retrieved across them, judge its own completeness, and generate an answer–with few constraints, little utilization of schemas, and text as the primary representation throughout. We argue that this methodology is an ineffective abstraction for enterprise data, and that reliable agentic AI should instead require structure: a constrained query interface over each source and a table-centric integration layer that a query processor drives. We introduce RUBICON, a system that embodies this vision.

RUBICON is based on two fundamental observations. First, text-to-SQL fails on real enterprise data and must be dramatically subsetted to achieve reliable results. Second, data integration across disparate corporate datasets is best performed using tables as the core abstraction rather than the text-centric pipelines of current LLMs and agentic AI systems.

We evaluate RUBICON on two benchmarks: on our own enterprise-focused benchmark, RUBICON-Bench, against two agentic systems, as well as on a recent semantic query benchmark called SemBench against two state-of-the-art semantic query processing systems, LOTUS and Palimpzest. On RUBICON-Bench, where queries require coordinating across heterogeneous enterprise sources, RUBICON achieves 100% end-to-end accuracy, while all agentic baselines — including single- and multi-agent ReAct systems — produce no correct answers. On SemBench, RUBICON surpasses both LOTUS and Palimpzest: it achieves 14.7% higher accuracy, reduces end-to-end latency by 62.64%, and lowers token cost by 98.64%, demonstrating that a table-centric architecture better matches the structure of enterprise data while also yielding significant efficiency gains in enterprise settings.

VLDB Workshop Reference Format:
Fabian Wenz[1,2] Felix Treutwein[4] Çağatay Demiralp[3,2] Michael Stonebraker[2] . RUBICON: Agentic AI for Messy Enterprise Data. VLDB 2026 Workshop: Applied AI for Database Systems and Applications (AIDB 2026).

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/fabian-wenz/rubicon.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

1 INTRODUCTION

Large Language Models (LLMs) were all the rage three years ago as researchers tried to figure out what to do with them. It quickly became apparent that LLMs needed to be augmented by RAG, fine tuning, prompt engineering, etc. to achieve good results. Hence, LLMs quickly evolved into agentic AI, a pipeline (or graph) of processing steps centered around an LLM. Agentic AI requires an orchestrator to decide on a sequence of actions to take, and traditional agentic AI entrusts this step to an LLM [2, 7].

We believe there are two fundamental problems with this architecture in an enterprise setting.

First, one needs to access much corporate data through SQL interfaces. Although LLMs do reasonably well on public text-to-SQL benchmarks, such as Spider [9] and Bird [5] where accuracies above 80% [6] are reported, they fail miserably on real-world-inspired benchmarks such as Beaver [1] (accuracy around 10%). Accuracy on Beaver rises to 30+% when the prompt includes the tables in the FROM clause and all the join terms. The reasons are detailed in [1] and include:

(1) **enterprise data is generally not available for training LLMs as it is proprietary:** An adage in [3] is that an LLM can't retrieve data unless it has seen it at least a couple of times before.
(2) **schema and technical debt ("schema rot"):** This includes opaque names for tables and columns, semantic overlap between tables, and redundancy (materialized views). This is an example of messy enterprise data.
(3) **idiosyncratic data:** For example, in the MIT data warehouse buildings have a number, not a name.
(4) **complex queries:** In real data warehouses the queries are more complex than those in Spider or Bird.

Hence, full SQL is nearly guaranteed to generate failure in an agentic AI environment. As a result, one requires "baby SQL," for which we propose the Agentic Query Language (AQL).

The second reason concerns the mechanism for data integration. Using text as the common representation for integration—that is, converting every data source into text and asking an LLM to combine the results—will cost a lot of money in tokens and is likely to produce poor accuracy. For example, an insurance underwriting application must integrate maps (say of flood zones), CAD (building drawings), previous claims (tables), and underwriting notes (text). It is very unlikely that converting all these sources to text so an LLM can do the integration will succeed. A much better idea is to convert the data sources to tables. This will require a table orchestrator, in other words, a relational query optimizer.

For these two reasons, we propose an alternate agentic AI architecture, well-suited to enterprise data. It leverages three main concepts:

AQL as the mechanism to access corporate data. This requires tabular orchestration to generate complex SQL commands, as well as adapters to access data behind bespoke interfaces.

Text access through LLMs, embeddings, or inverted indexes. The resulting documents are converted to tables and ingested by a query optimizer.

A human in the loop. Since searches may go "off the rails," the user must be able to inspect intermediate results and correct the plan. Human guidance is also required to orchestrate complex queries.

The remainder of this paper discusses AQL and the architecture of RUBICON. We then conclude with benchmark results comparing RUBICON to other approaches on a benchmark inspired by a real application on which we are collaborating with the Department of Transportation of the City of Munich (Germany).

2 AGENTIC QUERY LANGUAGE

RUBICON has three moving parts: AQL, wrappers, and intermediate results. AQL is the user-facing notation. Wrappers are the source-facing translation layer. Intermediate results are the system-facing representation that makes execution inspectable and optimizable.

The core abstraction is Agentic Query Language (AQL), a restricted SQL-like notation that is a compromise between natural language and SQL. Natural language is easy to write, but has unacceptably bad accuracy in enterprise settings. Full SQL has the opposite problem. It is explicit, optimizable, and reproducible, but requires users to know the schema and write precise queries. AQL is a compromise between these two extremes.

The basic AQL form is:

```
FIND <columns>
FROM <source or table>
WHERE <natural-language predicate>
```

The user specifies the data source, relation, and desired attributes explicitly; otherwise there is little hope for acceptable accuracy. The predicate may remain in natural language. AQL also supports different degrees of explicitness. For example, a novice may write:

```
FIND professors
FROM UniversityDW
WHERE faculty members affiliated with the
research lab
```

whereas an expert may write:

```
FIND Faculty.name, Faculty.title
FROM UniversityDW.Faculty
WHERE Faculty.affiliation = 'Research Lab'
AND Faculty.rank IN ('Professor', 'Full
Professor')
```

The attributes specified in the FIND clause are not required to exist in the source. Database wrappers may construct derived attributes using expressions supported by the native query language. LLM-derived attributes, however, require an LLM source whose output is joined with the original relation. For example, in a different query, the LLM may produce a Boolean attribute `likely_about_AI` for each course, which is then joined back into the course relation.

Both queries produce the same logical result object:

```
ResearchLabProfessors(name, title)
```

The difference is not the desired answer, but how much structure the user chooses to expose. The more structure is specified, the less work is delegated to the LLM or wrapper.

AQL also supports joins by composing two source-local queries:

```
FIND name, title
FROM UniversityDW.Faculty
WHERE faculty members affiliated with the
research lab
JOIN
FIND name, award
FROM Wikipedia.Pages
WHERE people who have won a Turing Award
or Nobel Prize
```

Here, the shared `name` attribute supplies the inferred join key. When exact equality is not specified, the join may use entity resolution to match corresponding records across sources.

The important point is not syntactic novelty; rather it is allocation of responsibility. The human or interface specifies the structural part of the query: sources, fields, and intermediate objects. The wrapper or LLM handles only the local ambiguity inside the predicate. This makes LLM use narrow, inspectable, and replaceable.

AQL also includes minimal schema-inspection and housekeeping commands. The command `?` lists available sources, `? <source>` lists available relations in a source, and `? <relation>` lists attributes. Intermediate results can be named using `SAVE`. These commands are deliberately mundane. Their purpose is to make exploration explicit rather than conversational.

As a result, a novice can start with mostly natural-language predicates and a GUI that helps select sources and fields, whereas an expert may write more explicit AQL approaching SQL. In both cases, the result is not a hidden chain of tool calls but a visible query object that can be saved, modified, re-executed, and optimized.

An unfamiliar user may initially spend some effort exploring the schema and data. However, a black-box system may require even more investigation if it misinterprets the user's intent, selects the wrong sources, or returns an incorrect result without revealing where the error occurred. AQL makes this process explicit, allowing the user to inspect and correct a particular source, field, predicate, or intermediate result rather than repeatedly rerunning an opaque end-to-end workflow.

Of course, it is possible to give RUBICON the entire user task and ask it to compile the task into an ordered collection of AQL commands. As mentioned earlier, we do not expect RUBICON to produce equally reliable results in this mode.

3 RETRIEVAL AS QUERY PROCESSING

RUBICON treats retrieval as a query processing problem. It is closer in spirit to mediator-wrapper data integration than to RAG-style ingestion: sources remain in place, wrappers expose logical relations, and integration occurs at query time.

Many enterprise copilots follow an ingestion-and-indexing architecture: they connect to enterprise systems, extract documents or records, compute embeddings or sparse indices, and expose a unified semantic search interface. This improves discoverability, but it does not integrate the underlying data models. A text index

is not a join engine. It is useful for finding evidence; it is not a substitute for an execution plan.

RUBICON instead follows the classical database approach: leave data sources in place, put a wrapper over each source, and perform integration at query time. A wrapper translates AQL into the source's native interface, relies on the source's existing execution and access-control mechanisms, and normalizes the result into a logical relation. Hence, RUBICON does not reimplement the underlying systems. Adding a source generally requires only a one-time adapter that maps the relevant AQL operations to the source's API or query language and converts its output into the common tabular representation. The adapter can then be reused across queries. The source may be a database, a search engine, an e-mail system, a web API, or an LLM; the query processor sees each of them through the same logical interface.

For example, an e-mail system can expose a logical table:

```
Message(from, to, subject, date, body, thread_id)
```

A Wikipedia wrapper can expose:

```
Page(title, url, snippet, categories, text)
```

A data warehouse already exposes relations, while an LLM wrapper can expose its answer as a document relation with text, metadata, and provenance. These tables need not be materialized relations. They are logical views constructed by wrappers. This normalization step is essential: without a tabular representation, downstream operators such as joins, unions, and filters will fail.

Execution proceeds in two modes. In interactive mode, the user issues one AQL command, inspects the result, and then proceeds. Each command materializes an intermediate relation that can be inspected, saved, reused, or joined with later results. This mode is appropriate for exploratory enterprise questions, where the right plan is often discovered step by step.

In compiled mode, a sequence of AQL commands is executed as a single composite plan. Once the system has the full command sequence, it can apply database-style optimization: predicate pushdown, join reordering, source selection, and cost-aware execution. Costs may include not only runtime, but also tool calls, latency, and token consumption. In this sense, retrieval is not the step before query processing. Retrieval is one operator inside query processing.

In the current prototype, source selection and join order follow the user-specified command sequence. Source-local optimization, including filter pushdown and query planning, is delegated largely to the wrapped systems before their results are materialized as intermediate relations. The system records source calls, latency, and token consumption, but cost-based optimization across heterogeneous sources remains future work.

In this sense, retrieval is not a step before query processing; it is one operator within query processing.

4 RUBICON-BENCH AND INITIAL RESULTS

We have been working with the Department of Transportation of the City of Munich (Germany). More and more traffic engineers have to spend a significant amount of time answering citizen complaints/queries, often of the form "I don't have time to cross the street before the light turns" or "I have to wait too long before passing the traffic light at X/Y!" These engineers consult the following data sources to address such concerns: light schedule (tables), public transport schedule (tables), diagrams of intersections (CAD), past complaints, and normative context (German laws and Munich specific laws, technical guidelines, etc.). They have tried agentic AI with helpful qualitative results in simple research tasks on guidelines, but poor results with respect to reproducibility and unsatisfying options regarding the integration of aforementioned sources. They have found the preliminary RUBICON results encouraging. Unfortunately, it will be a long process to get permission to anonymize and make public their data and internal benchmarks.

In the meantime, we have constructed RUBICON-Bench, which is similar to the Munich problem. This benchmark models a university-style enterprise environment in which relevant information is distributed across five heterogeneous sources: an institutional data warehouse, a research laboratory website, Wikipedia/API access, e-mail, and LLM knowledge. The data warehouse contains structured administrative information such as people, buildings, rooms, and organizational affiliations. The lab website contains semi-structured public information about research groups, events, projects, and personnel. E-mail represents private communication, while Wikipedia and the LLM source represent external public knowledge.

The workload contains seven natural-language queries. Each query requires coordination across two relevant sources, while the remaining sources act as distractors. Representative queries include:

> *Which CSAIL professors have won a Turing Award or a Nobel Prize?*
> *What research lab events have taken place in a specific campus building over the past month?*

These queries are deliberately simple to read but nontrivial to execute. For example, the question *How many university buildings have a Wikipedia page?* requires retrieving the set of buildings from the university data warehouse, retrieving candidate pages from Wikipedia, resolving name variants, and counting the joined result. Similarly, the question *Which research lab professors have won a Turing Award or a Nobel Prize?* requires identifying research lab professors from the institutional source and joining them with external award evidence. In both cases the system must use the correct sources and make the cross-source join explicit.

Table 1: RUBICON-Bench results. Metrics are averaged over queries: $\overline{T}$ = tokens, $\overline{k}$ = calls, $\overline{C}$ = cost, Latency = seconds, $Acc1$ = answer accuracy, and $Acc2$ = source accuracy.

System	Query	$\overline{T}$	$\overline{k}$	$\overline{C}$ ($)	Latency(s)	$Acc1$	$Acc2$
Vanilla	NL	1 105.57	0.0	0.0016	13.70	0.00%	0.00%
Vanilla	AQL	1 447.43	0.0	0.0021	12.80	0.00%	0.00%
Single-Agent	NL	7 609.43	2.14	0.0180	16.07	0.00%	0.00%
Single-Agent	AQL	36 986.00	4.57	0.0754	25.75	0.00%	71.43%
Multi-Agent	NL	34 772.43	10.00	0.0800	47.94	0.00%	14.29%
Multi-Agent	AQL	8 672.14	3.14	0.0324	28.72	28.57%	100.00%
RUBICON	NL	2 440.29	2.00	0.0048	6.99	100.00%	100.00%
RUBICON	AQL	242.29	2.00	0.0011	4.37	100.00%	100.00%

Note. All runs in this table use OpenAI's GPT-5.2.

Table 1 reports the results of running RUBICON-Bench on the execution modes considered in this paper. Each of four systems is run in two different modes. In the natural-language (NL) variant,

the system is given only the user question and must decide for itself which sources to access and how to combine them. In the AQL-guided variant, the input is an explicit query plan that names the sources and specifies the intended multi-source execution plan.

The first system is a vanilla LLM (OpenAI's GPT-5.2), and the two variants are supported through prompt engineering. The next rows show a Single-Agent ReAct [8] system which is a single LangChain[1] ReAct agent with five tools (SQLite, Gmail, Wikipedia, GPT, and CSAIL). Given a natural-language question, the agent freely decides which tools to call and in what order through its own reasoning loop. The AQL-guided variant keeps the exact same agent and ReAct loop but reformats the input as an explicit AQL JOIN plan (naming the two sources and what to retrieve from each).

The third set of rows shows the Multi-Agent variant, namely a LangGraph[2] StateGraph where the orchestrator itself is the ReAct-loop[8]. It reasons ("what do I need next?"), acts (calls one sub-agent), observes the result (inspects collected results), and reasons again before deciding the next step — up to four times.

Each sub-agent is a single-tool ReAct loop restricted to one data source. The AQL-guided variant uses the same graph but passes the pre-written AQL plan as an instruction, telling the orchestrator exactly which two agents to call and in what order, effectively constraining its iterative reasoning to match the pre-specified plan.

The last option is RUBICON. It is evaluated in the same two input modes used throughout the paper. As explained in Section 2, NL mode corresponds to a novice natural-language request, while AQL mode corresponds to a more explicit expert specification.

We report two accuracy metrics in Table 1. $Acc1$ is end-to-end execution accuracy. A query receives score 1 only if the final answer is correct and 0 otherwise. There is no partial credit for calling one of the right sources or making progress toward the answer. On this metric, only RUBICON performs well.

$Acc2$ is a source-selection metric. It asks a simpler question: did the system call the correct sources, tools, or agents? The AQL variants do well on $Acc2$, largely by construction, since the plan specifies the required sources. The NL variants do poorly because the LLM planner must infer the source set from the question alone.

The remaining columns report aggregate efficiency metrics averaged over the benchmark. $\overline{T}$ is the mean number of tokens per query, $\overline{k}$ is the mean number of tool calls, $\overline{C}$ is the mean provider-reported cost, and latency is the mean time to first answer. These metrics show that RUBICON has lower latency than the agentic baselines and is generally a great deal cheaper.

The results support several observations. Vanilla LLM scores 0% on all queries in both modes. Even an explicit AQL sequence does not help. Its single one-shot operation naturally causes low token numbers, cost, and latency. On the other hand, agentic methods result in each additional reasoning step is another LLM call, thereby increasing token consumption, cost, and latency.

Among the agentic baselines, the Single-Agent NL follows a narrow linear chain. It tries to find the answer by successively narrowing the scope, which keeps cost low but usually misses a second data source, if one exists. Of course, Single-Agent AQL fixes the source-selection problem by specifying both sources explicitly. The price is two full ReAct loops plus a synthesis step, which yields roughly five times as many tokens. A Multi-Agent NL system has an orchestrator that sees the full source landscape and explores broadly, exhausting its call budget on every query and producing the highest token count and cost. Multi-Agent AQL restricts execution to the required sources, cutting both tokens and latency.

$Acc1$ remains near zero for all agent variants. The reason is simple: the sub-agents return prose, not relations. Hence the final "join" is a join over unstructured text, which language models do unreliably. RUBICON, on the other hand, retrieves relational tables and applies explicit joins. As a result, RUBICON is the clear winner across all metrics. Its AQL variant uses the fewest tokens, completes fastest, and achieves 100% on both $Acc1$ and $Acc2$. The NL variant is, of course, more expensive because of the cost of NL. Even so, it is still cheaper and faster than any agentic baseline, while achieving perfect accuracy.

We also ran RUBICON on SemBench [4], another multimodal query scenario of similar complexity to RUBICON-Bench. Notice that RUBICON-Bench is focused on whether a system can identify the correct data sources, join their outputs, and determine that the answer is complete. In contrast, SemBench tests cost and accuracy of fixed pre-specified plans.

We evaluated the SemBench E-Commerce scenario. The workload is one logical source, a fashion dataset, exposed through three access paths: structured attributes, text embeddings, and image embeddings. The joins are therefore intra-catalog joins, not joins across independent enterprise systems. As a result, it is much simpler than Rubicon-Bench.

We executed the workload in the intended RUBICON style: source-local FIND operations produce intermediate tables, and subsequent operators join these tables explicitly. This simple design works well. RUBICON achieves an average quality of 0.86 at a cost of $0.003 and latency of 55.74 s. On the same scenario, SemBench reports 0.75 quality, $0.22 cost, and 149.2 s latency for LOTUS, and 0.70 quality, $0.42 cost, and 192.5 s latency for Palimpzest [4]. For this single-source, multimodal workload, treating semantic predicates as explicit operators and joining materialized intermediate results is cheaper and more robust than pushing the whole task through an LLM-driven execution engine.

5 CONCLUSION

This work has evaluated RUBICON against traditional LLMs and agentic AI. The basic question is whether a less constrained interface can perform well using increased reasoning effort and broader tool access on multi-source coordination retrieval problems. Across all configurations, a consistent pattern emerges: greater autonomy does not translate into correct multi-source integration. Even when equipped with full tool access and high reasoning effort, models frequently fail to identify all required sources, invoke them in the appropriate order, or enforce completeness before termination.

Unless (or until) this problem is rectified, one must use a tool like RUBICON. In addition, one can imagine using a hybrid system where all structured data sources are serviced by something like RUBICON. Then traditional agentic AI would perform the integration between text and tables. As a next step, we will compare such a system with RUBICON, but we are skeptical that it will work better. Discarding

[1] https://github.com/langchain-ai/langchain
[2] https://github.com/langchain-ai/langgraph

the structure in structured data so it can be combined with text in a text-oriented system just throws away semantics. It is hard to believe that this is a good idea. Furthermore, entrusting integration to an agentic AI system increases token consumption and increases cost.

Overall, the results indicate that LLM-centric coordination does not work well on enterprise-grade multi-source problems. By contrast, RUBICON provides more robust results through its human-in-the-loop, database-oriented architecture.

REFERENCES

[1] Peter Baile Chen, Fabian Wenz, Yi Zhang, Moe Kayali, Nesime Tatbul, Michael J. Cafarella, Çagatay Demiralp, and Michael Stonebraker. 2024. BEAVER: An Enterprise Benchmark for Text-to-SQL. *CoRR* abs/2409.02038 (2024). https://doi.org/10.48550/arXiv.2409.02038

[2] Timo Eckmann and Carsten Binnig. 2026. A Vision for Autonomous Data Agent Collaboration: From Query-by-Integration to Query-by-Collaboration. In *16th Conference on Innovative Data Systems Research*. www.cidrdb.org. https://vldb.org/cidrdb/2026/a-vision-for-autonomous-data-agent-collaboration-from-query-by-integration-to-query-by-collaboration.html

[3] Nikhil Kandpal, Haikang Deng, Adam Roberts, Eric Wallace, and Colin Raffel. 2023. Large language models struggle to learn long-tail knowledge *(ICML'23)*. JMLR.org, Article 641, 12 pages. https://doi.org/10.5555/3618408.3619049

[4] Jiale Lao, Andreas Zimmerer, Olga Ovcharenko, Tianji Cong, Matthew Russo, Gerardo Vitagliano, Michael Cochez, Fatma Özcan, Gautam Gupta, Thibaud Hottelier, H. V. Jagadish, Kris Kissel, Sebastian Schelter, Andreas Kipf, and Immanuel Trummer. 2025. SemBench: A Benchmark for Semantic Query Processing Engines. *CoRR* abs/2511.01716 (2025). https://doi.org/10.48550/ARXIV.2511.01716

[5] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024. Can LLM Already Serve as a Database Interface? A Big Bench for Large-Scale Database Grounded Text-to-SQLs. *Advances in Neural Information Processing Systems* 36 (2024).

[6] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems* 36 (2024).

[7] Matthias Urban and Carsten Binnig. 2024. Demonstrating CAESURA: Language Models as Multi-Modal Query Planners. In *Companion of the 2024 International Conference on Management of Data*. ACM, 472–475. https://doi.org/10.1145/3626246.3654732

[8] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations*. OpenReview.net. https://openreview.net/forum?id=WE_vluYUL-X

[9] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. *arXiv preprint arXiv:1809.08887* (2018).