UNIVERSITY OF WATERLOO
FACULTY OF MATHEMATICS
David R. Cheriton School
of Computer Science

LLM–DBMS Semantic Processing: Comparative Evaluation

DocETL (Agentic) • Lotus (Algebraic) • Palimpsest (Rewriting) • ELEET (ML-based)

Prepared by: Mazen El Sedfy — September 11, 2025

Led by: Kerem Akillioglu - Supervised by: Tamer Ozsu

Abstract

This report benchmarks three LLM-powered data processing approaches for SQL-shaped workloads using enterprise GPT-4o: Lotus (algebraic operators), DocETL (agentic orchestration), and Palimpsest (prompt rewriting). Lotus delivered the lowest latency and cost on all five query shapes (Projection, Filter, Multi-LLM, Aggregation, RAG). DocETL's orchestration overhead raised latency and cost, and several documented conveniences did not hold in my environment. Palimpsest improved robustness via rewriting/selection but was significantly more expensive due to extra calls and tokens. ELEET (ML-based rewriting) was excluded from head-to-head results due to setup hurdles and mismatched claims.

One-line takeaway: Use Lotus by default; add DocETL when guardrails/agentic control are mandatory; apply Palimpsest surgically to "hard" rows under strict budgets.

1. Introduction & Scope

Goal: Evaluate LLM integration patterns for DBMS-like semantics over tabular and RAG workloads, with emphasis on operator cost/latency, reliability, and developer ergonomics.

Scope: Enterprise inference only (OpenAI GPT-4o). No local model inference was used in these measurements.

1.1 Terminology & Query Shapes

• Q1 Projection — LLM in SELECT to synthesize recommendations from {movie_info, review_content}.

• Q2 Filter — LLM in WHERE to classify kid-suitability (restrict to Fresh reviews).

• Q3 Multi-LLM — SELECT recommendations + WHERE kid-suitability (two LLM calls per row).

• Q4 Aggregation — LLM rates satisfaction 0–5 per review; AVG by title.

• Q5 RAG — Retrieve context via similarity search; LLM answers using the provided context.

2. Systems Overview (Claims vs. Observed)

2.1 Lotus — Algebraic Operators

Observed: Natural mapping to SQL-shaped logic; minimal orchestration overhead; strong reliability; simplest batching and memoization. Best latency/cost in all queries.

2.2 DocETL — Agentic Orchestration

Observed: Quick prototyping of multi-step flows with guardrails. However, several advertised conveniences did not hold in my environment; executing SQL-like workloads required extra glue code and was difficult. Overhead from plan/validate loops increased latency and cost.

2.3 Palimpsest — Prompt Rewriting & Selection

Observed: Rewriting/selection improved robustness on messy inputs, but the extra rewrite(k)/selection/final calls significantly increased tokens and cost; latency also rose.

2.4 ELEET — ML-based Rewriting

Observed: Setup was difficult and claims of easy integration did not match my results, preventing fair inclusion in the head-to-head numbers. Concept remains promising for learned rewrite policies.

3. Evaluation Framework & Methodology

3.1 Experimental Setup

• Inference: Enterprise GPT-4o across all included systems for apples-to-apples comparisons.

• Measurement: wall-clock latency per query; API billing for cost; identical input rows per query; consistent output schemas.

• Fairness controls: same top-k and chunking for RAG; fixed max_tokens and temperature per query; stable JSON schemas.

3.2 Datasets & Workloads

• Rotten Tomatoes (movies + reviews) for Q1–Q4 (1000 samples per query).

• SQuAD subset for Q5 (RAG) (1000 samples per query).

3.3 Cost Accounting Assumptions

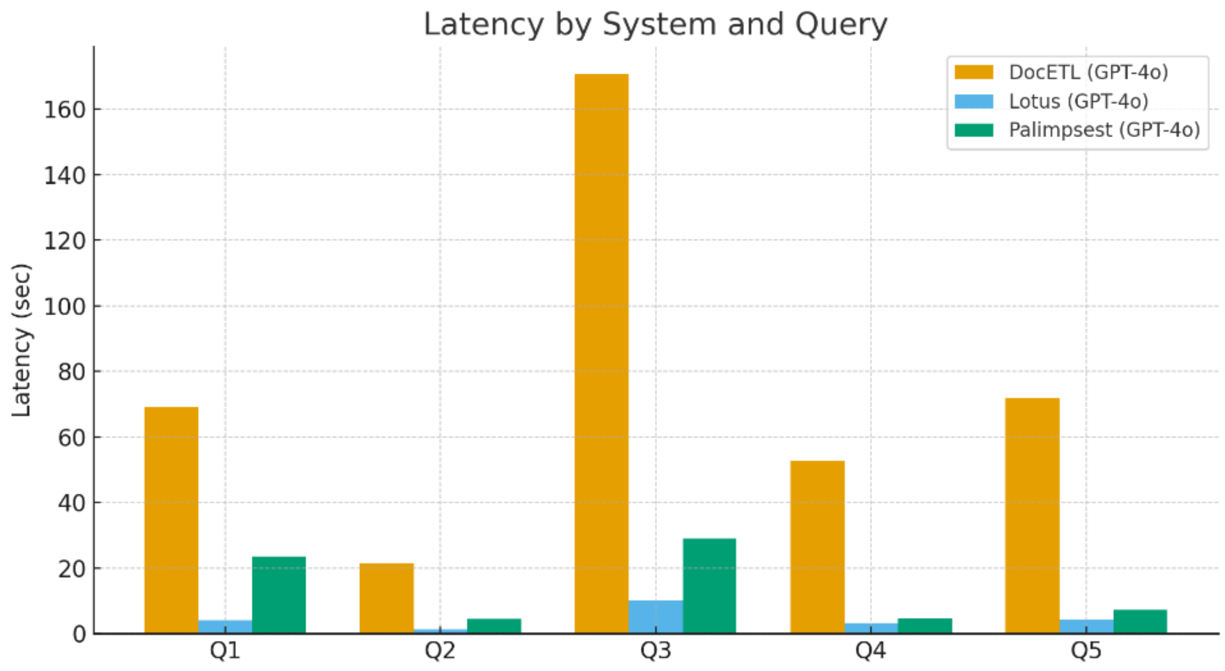
Costs include every LLM call a system initiates (planning, validation, rewriting, selection, and the final call). Units are USD. The Palimpsest Q4 spacing anomaly ('$ 0.85') was normalized to $0.85 for charting.

4. Results

4.1 Latency (seconds)

System \ Query	Q1	Q2	Q3	Q4	Q5
DocETL (GPT-4o)	69.07	21.53	170.60	52.68	71.75
Lotus (GPT-4o)	4.13	1.27	10.20	3.15	4.29
Palimpsest (GPT-4o)	23.48	4.50	29.14	4.64	7.29

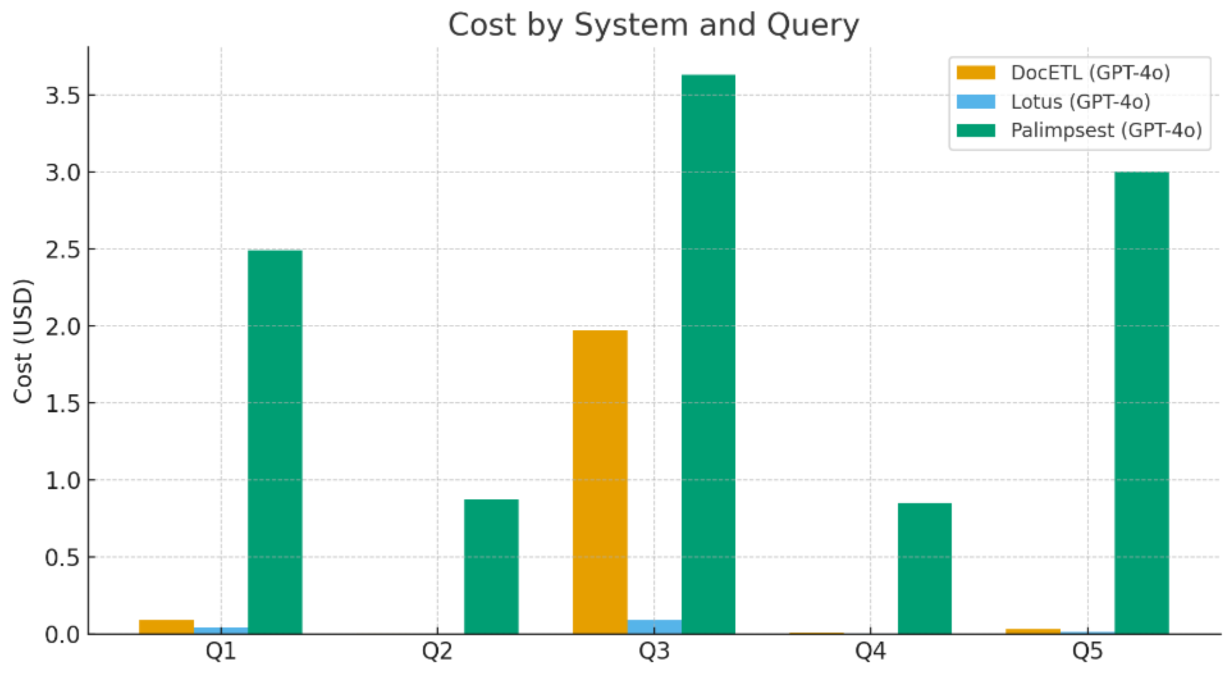
Average latency (sec): DocETL (GPT-4o): 77.13, Lotus (GPT-4o): 4.61, Palimpsest (GPT-4o): 13.81



4.2 Cost (USD)

System \ Query	Q1	Q2	Q3	Q4	Q5	Avg	Sum
DocETL (GPT-4o)	$0.090000	$0.004844	$1.971428	$0.010266	$0.032143	$0.421736	$2.108681
Lotus (GPT-4o)	$0.042080	$0.002265	$0.091975	$0.004770	$0.014908	$0.031200	$0.155998
Palimpsest (GPT-4o)	$2.490000	$0.875000	$3.630000	$0.850000	$2.998000	$2.168600	$10.843000

Note: Palimpsest costs were significantly higher due to rewrite/selection passes. Q4 spacing anomaly normalized to $0.85 for consistency.



5. Analysis

5.1 Why Prices Differ (Same Logical Query)

Cost ≈ (number of LLM calls) × (tokens_in + tokens_out) × (price per token).

Drivers: (A) extra calls from planning/validation or rewrite/selection; (B) longer prompts (system shells, schemas, meta-instructions); (C) retries/guardrails; (D) RAG context size; (E) output schema verbosity.

5.2 Typical Call Counts per Row

Query	Lotus	DocETL	Palimpsest
Q1 Projection	1	2–3 (plan+exec[+validate])	4–5 (k rewrites + select + final)
Q2 Filter	1	3 (plan+exec+validate)	4–5
Q3 Multi-LLM	2	4–6	7–9
Q4 Aggregation	1 per row	2–3 per row	4–5 per row
Q5 RAG	1 (generate)	2–3 (plan/QA)	4–5 (rewrite+select+final)

5.3 Operator Differences by System

Aspect	Lotus (Algebra)	DocETL (Agentic)	Palimpsest (Rewriting)
Core model	Operators: SELECT/WHERE/AGG/RAG	Graph: plan→execute→validate (+tools)	Pipeline: REWRITE(k)→SELECT→FINAL
Composition	Declarative; vectorizable	Imperative: control-flow & retries	Wraps a base call; augments prompts
Batching	Native, stable	Possible; limited by step state/rate limits	Possible; k× selection multiplies tokens
Caching	Per-operator/row hash	Per-step (plan/exec/validate)	Rewrite cache by query shape + traits
Schema enforcement	Tight prompts/JSON contracts	Post-hoc validation/repair	Better prompts via rewrites; final still must obey
RAG fit	Clean: retrieve→generate	Add retrieval QA/fallbacks	Rewrite instruction for robustness
Cost profile	Lowest calls/tokens	Medium–high	Highest (k + selection + final)

6. Engineering Experience & Operational Notes

• Setup: ELEET and DocETL setup was difficult; several claims did not hold in practice. SQL-like execution was hard in DocETL without custom glue.

• Reliability: Lotus matched docs with observed behavior; easiest to reason about and operate.

• Instrumentation: Trace latency, tokens, retries, cache hits; enforce JSON schema to reduce re-asks.

• Cost control: Cap candidates/tokens (Palimpsest), disable planning for trivial cases (DocETL), and standardize prompt shells across systems.

7. Recommendations

• Default to Lotus for Projection/Filter/Aggregation/RAG on tables.

• Reserve Palimpsest for hard/heterogeneous rows with k≤2 and strict budgets.

• Use DocETL where agentic control and guardrails are mandatory; accept overhead.

• Normalize RAG top-k/chunking and cap max_tokens; unify JSON schemas across systems.

• Implement operator-level caching with TTL and invalidation; measure hit-rates.

8. Impact on Kerem's PhD & Future Work

This study yields empirical cost/latency profiles for LLM operators, documents failure modes and claim-vs-reality gaps for agentic/rewrite systems, and provides a clear baseline where Lotus-style algebra dominates for SQL-shaped workloads—informing optimizer design, operator interfaces, and runtime policies for an LLM-aware DBMS.

Roadmap highlights: (1) operator-level caching; (2) token-aware cost models and plan enumeration; (3) adaptive routing (send only "hard" rows to Palimpsest); (4) provenance and validation operators; (5) benchmark pack for reproducibility; (6) learned rewrite policies (ELEET) once setup stabilizes.

9. Conclusion

Across five canonical query shapes on enterprise GPT-4o, Lotus was the most reliable, easiest to build on, and most efficient. DocETL provided agentic control but incurred overhead and did not meet some claimed conveniences in my environment, especially for SQL-like execution. Palimpsest offered robustness via rewriting at a significantly higher cost. For DBMS-style semantics, the algebraic operator approach is the pragmatic default, complemented by agentic and rewriting techniques where they add specific value.

Appendix A — Query Definitions

```
Q1 — Projection:
SELECT LLM("Recommend movies ...", m.movie_info, r.review_content)
FROM reviews r JOIN movies m ON r.rotten_tomatoes_link = m.rotten_tomatoes_link

Q2 — Filter:
SELECT m.movie_title ... WHERE LLM("Analyze kid suitability ...", m.movie_info, r.review_content)
== "Yes" AND r.review_type == "Fresh"

Q3 — Multi-LLM:
SELECT LLM("Recommend ...", m.movie_info, r.review_content) AS recommendations ... WHERE
LLM("Analyze kid suitability ...") == "Yes" AND r.review_type == "Fresh"

Q4 — Aggregation:
SELECT AVG(LLM("Rate 0-5 ...", r.review_content, m.movie_info)) AS AverageScore ... GROUP BY
m.movie_title

Q5 — RAG:
SELECT LLM("Given {context}, answer this question", VectorDB.similarity_search(s.question),
s.question)
FROM squad s WHERE s.is_impossible = False
```

Appendix B — Side-by-Side Prompt Templates

Minimal, parity-focused shells (trimmed).

Q1 — Projection

```
Lotus:
System: concise recommender; return JSON.
User: TASK ...
INPUT: movie_info, review_content
OUTPUT SCHEMA: {"recommendations": ["<title>"]}

DocETL:
System: ETL agent (plan→execute→validate)
PLAN: minimal steps; schema
EXECUTE: movie_info, review_content → JSON

Palimpsest:
REWRITE k=3 → SELECT best → FINAL with inputs; return JSON only.
```

Q2 — Filter

```
Lotus:
System: strict classifier. Output exactly Yes or No.
User: Is this kid-suitable? ...
```

```
DocETL:
PLAN: single classification; schema {"is_kid_safe":"Yes|No"}
EXECUTE: inputs → schema; VALIDATE if mismatch.

Palimpsest:
REWRITE k=2 → SELECT → FINAL; force Yes/No.
```

Q5 — RAG

```
Lotus:
System: use only provided context; cite IDs; return {answer, citations}.

DocETL:
PLAN: verify retrieval, answer with schema, validate citations.
EXECUTE: context + question → JSON.

Palimpsest:
REWRITE k=3 for anti-hallucination; SELECT; FINAL with context; return {answer, citations}.
```