BatchDAG: LLM-Planned Execution Graphs for Scalable Ad-Hoc Analysis Over Enterprise Data

Anupreet Walia
Brevian.ai
anu@brevian.ai

Abstract

Large language models (LLMs) excel at analyzing individual documents but break down when users ask exhaustive, cross-entity analytical questions over enterprise-scale datasets—for example, "Did our account executives open every meeting with effective discovery questions?" across 50,000 recorded meetings. The single-agent tool-calling paradigm fails at this scale for three compounding reasons: context window overflow, loss of per-entity attribution in global top-N retrieval, and linear wall-clock time growth from sequential tool calls. We present **BatchDAG**, a system in which an LLM generates a typed directed acyclic graph (DAG) of operations—SQL queries, semantic searches, in-memory transforms, parallel fan-outs, and single-shot analyses—which a deterministic execution engine then evaluates with topological-wave parallelism. Structured JSON rows flow between steps (never prose summaries), enabling proper joins, filters, and grouping. A key optimization, *entity-aware batching*, groups input rows by their logical entity before fan-out, reducing LLM calls by up to 47× while preserving per-entity attribution. BatchDAG has been deployed in production at Brevian.ai, where it processes analytical queries over corpora of 50,000+ meetings and 3,000+ sales opportunities in under 60 seconds, with measured per-query costs ranging from ~$0.02 for SQL-only queries to ~$0.24 median for transcript-heavy fan-out analyses, computed from actual GPT-5.1 token counts at published API pricing. BatchDAG is not primarily an accuracy improvement over hand-optimized pipelines; rather, it is a general-purpose orchestration layer that replaces multiple hand-engineered pipelines with a single system that generates the appropriate execution strategy from natural language. In controlled experiments, BatchDAG (3.74/5) achieves quality comparable to an expert-designed pipeline (3.25/5) while eliminating the need to design such pipelines manually, and significantly outperforms a ReAct agent (3.09/5, $p<0.01$), while providing superior provenance (77% transcript evidence rate vs. 46–60% for baselines). A controlled ablation further shows that structured JSON intermediates—a core design choice—reduce hallucinations by 27% compared to prose summaries (paired t-test, $p=0.107$, $n=12$), suggesting that intermediate representation, not just model capability, is important for reliable multi-step reasoning. The planner achieves 98.8% valid-DAG rate across 300 planning calls.

1. Introduction

Tool-augmented LLM agents have become the standard architecture for enterprise question-answering systems. In this paradigm, an LLM reasons about a user's question, selects tools (database queries, semantic search, API calls), processes the results, and generates an answer. Systems such as ReAct, Toolformer, and commercial implementations in Langchain and LlamaIndex have demonstrated strong performance on queries that target individual entities or small document sets.

However, enterprise users increasingly ask a different class of question: *exhaustive, cross-entity analytical queries* that require processing hundreds to thousands of entities, each needing its own retrieval, contextual analysis, and per-entity attribution. Examples include:

- "Did account executives start meetings with good discovery questions?" (requires analyzing transcripts of every meeting)

- "Analyze every deal to see if security insurance was covered" (requires per-deal transcript search and classification)
- "For each meeting, check if the key stakeholder negotiated on price" (requires entity-level attribution across 3,000+ deals)

These queries expose three fundamental limitations of the single-agent loop:

1. **Context window overflow.** 50,000 meetings × 25 top transcript results = 1.25M tokens, exceeding even the largest commercial context windows.
2. **Loss of per-entity attribution.** Global top-N search returns the N most relevant results across ALL entities. The system can report that a topic was discussed somewhere, but cannot attribute findings to specific deals.
3. **Linear wall-clock time.** Sequential tool calls mean latency grows proportionally with entity count, making large-corpus analysis impractically slow.

We present BatchDAG, a system that addresses these limitations by decomposing the problem into two phases: (1) an LLM *planner* that generates a typed DAG of operations from a natural-language query, and (2) a deterministic *execution engine* that evaluates the DAG with topological-wave parallelism, structured data flow, and entity-aware batching. The key insight is that most analytical queries can be decomposed into a small set of typed operations (SQL, search, transform, fan-out, analyze, compare), most of which require zero LLM calls during execution. Only fan-out—the operation that applies LLM reasoning per entity batch—incurs LLM cost, and entity-aware batching minimizes that cost dramatically.

BatchDAG's primary contribution is not improving the quality of any single pipeline, but eliminating the need to design pipelines at all—replacing a collection of hand-engineered workflows with a single system that generates the appropriate execution strategy from natural language. In contrast to prior work that improves individual pipelines or agent reasoning, BatchDAG addresses a different problem: automatically selecting and composing the correct execution pipeline for each query from natural language.

Our contributions are:

1. A typed DAG formalism for decomposing ad-hoc analytical queries into composable operations with structured inter-step data flow.
2. An entity-aware batching algorithm that groups rows by logical entity before fan-out, achieving up to 47× reduction in LLM calls compared to row-level batching.
3. A goal-based planning prompt architecture that outperforms both exhaustive-rules and few-shot examples for generating correct DAGs.
4. A production deployment report covering cost, latency, and correctness over enterprise-scale data (50K+ meetings, 3K+ opportunities).
5. A controlled empirical evaluation demonstrating that automatically generated DAG pipelines achieve quality comparable to expert-designed baselines, with superior provenance (77% transcript evidence rate) and 27% fewer hallucinations through structured intermediates.

2. Related Work

2.1 Tool-Augmented LLM Agents

ReAct (Yao et al., 2023) interleaves reasoning and action steps in a single LLM loop. Toolformer (Schick et al., 2023) fine-tunes language models to insert API calls. Both operate on single queries and do not address the entity-level parallelism required for cross-corpus analysis. Commercial frameworks including LangChain, LlamaIndex, and Semantic Kernel provide tool-calling abstractions but delegate execution planning to the LLM at each step, inheriting the sequential bottleneck.

2.2 Query Planning and Decomposition

Least-to-Most prompting (Zhou et al., 2023) decomposes complex questions into sub-questions but generates natural-language intermediate results, losing structured composability. Plan-and-Solve (Wang et al., 2023) generates multi-step plans but executes them sequentially within a single context. Self-Discover (Zhou et al., 2024) selects reasoning modules for task composition but targets single-instance reasoning rather than data-parallel analysis. SQL generation approaches (DIN-SQL, DAIL-SQL) handle structured queries but cannot combine SQL with unstructured retrieval and LLM-based analysis in the same execution graph.

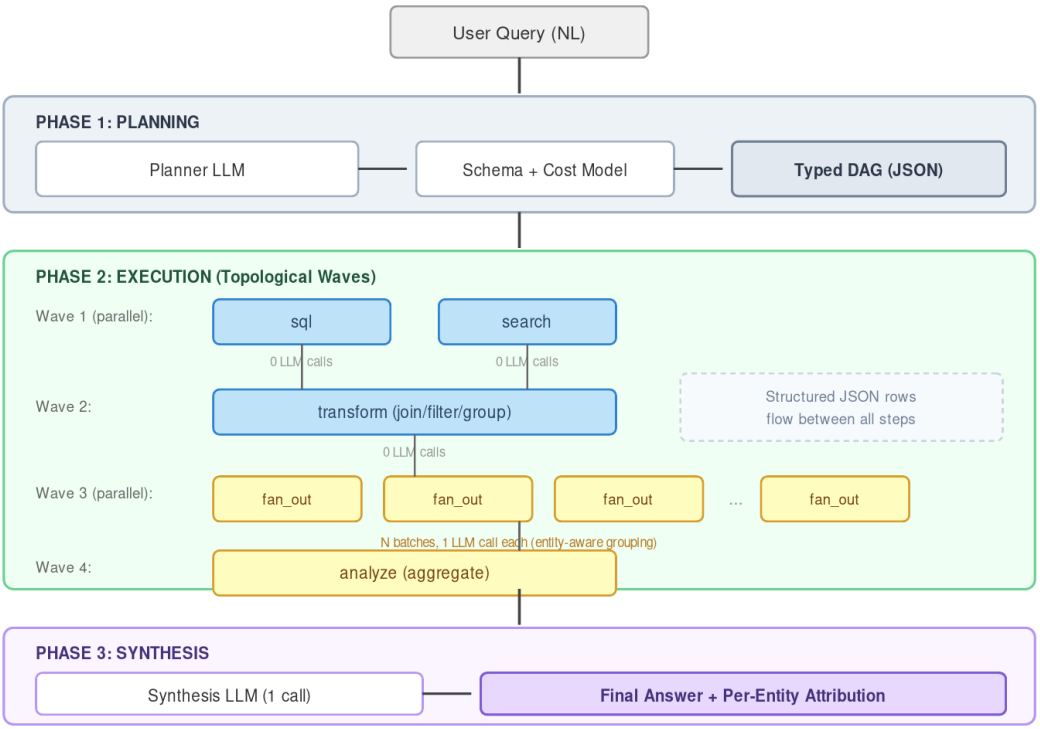
2.3 Map-Reduce for LLM Workloads

Map-Reduce patterns have been applied to LLM summarization (Langchain MapReduceChain) and document analysis. However, these approaches use fixed two-stage pipelines (map all documents, then reduce) and do not support the heterogeneous operation graphs (SQL → search → transform → fan-out → analyze) required for complex analytical queries. BatchDAG generalizes the Map-Reduce pattern by allowing arbitrary DAG structures with typed operations and structured data flow between steps.

3. System Design

BatchDAG operates in three phases: planning, execution, and synthesis. Figure 1 illustrates the overall architecture.

Figure 1: BatchDAG System Architecture



3.1 Architecture Overview

A user query enters the system and initiates a BatchDAG workflow[1] on a durable execution engine. The workflow proceeds through three phases:

Phase 1 (Planning): A planner LLM receives the user query along with a schema description of available operations and data sources. It generates a typed DAG of steps, each with explicit input specifications declaring which prior step's output to consume and how to join or filter it.

Phase 2 (Execution): The execution engine performs a topological sort of the DAG and executes steps in waves. Steps within the same wave (no mutual dependencies) execute in parallel. SQL, search, transform, and compare steps are deterministic and require zero LLM calls. Fan-out steps spawn parallel child tasks that process entity batches concurrently, each batch receiving a focused LLM call. Analyze steps make a single LLM call on aggregated data.

Phase 3 (Synthesis): A synthesis LLM compiles all step results into a coherent final answer with per-entity citations and attribution.

3.2 Typed Operation DAG

Each step in the DAG has a type drawn from a fixed set of six operations. Table 1 summarizes the step types, their computational backends, and their LLM cost.

Table 1: BatchDAG Step Types

Type	Description	LLM Calls	Backend
sql	Parameterized query against relational database	0	Relational DB (SQL)
search	Semantic/keyword search with two-stage retrieval and reranking	0	Vector search engine + cross-encoder reranker
transform	In-memory data manipulation: filter, group, sort, join, pivot, window, dedupe, select	0	Pure Python
fan_out	LLM analysis prompt applied per entity batch with entity-aware grouping	N (1/batch)	LLM API
analyze	Single LLM call over aggregated data from prior steps	1	LLM API
compare	Differential analysis between two datasets	0	Pure Python

The critical design decision is that four of six step types (sql, search, transform, compare) require **zero LLM calls** during execution. Only fan_out and analyze invoke the LLM, and analyze is always a single call. This means that for queries answerable by SQL + transform + analyze, the total LLM cost after planning is exactly one call.

[1]We use a durable workflow execution engine for orchestrating long-running, fault-tolerant workflows with per-step retry semantics.

3.3 Structured Inter-Step Data Flow

Steps pass **structured JSON rows** between them, never prose summaries. This is the single most important architectural decision in BatchDAG. Each step declares its input via an InputSpec:

- **Direct reference:** {step: 1, key: "meeting_id"} extracts a specific column from a prior step's output.
- **Merge join:** {merge: [{step: 1, key: "id"}, {step: 3, key: "meeting_id"}]} performs a left-join between two prior steps.
- **Dependency resolution:** depends_on: [1, 2] auto-resolves from the first dependency with available data.

When we initially allowed the LLM to summarize intermediate results in natural language, downstream steps hallucinated data and lost attribution. Structured rows are less expressive but fully composable: they support real database-style joins, filters, and grouping between steps, and they preserve the provenance chain from source data to final answer.

3.4 Entity-Aware Batching

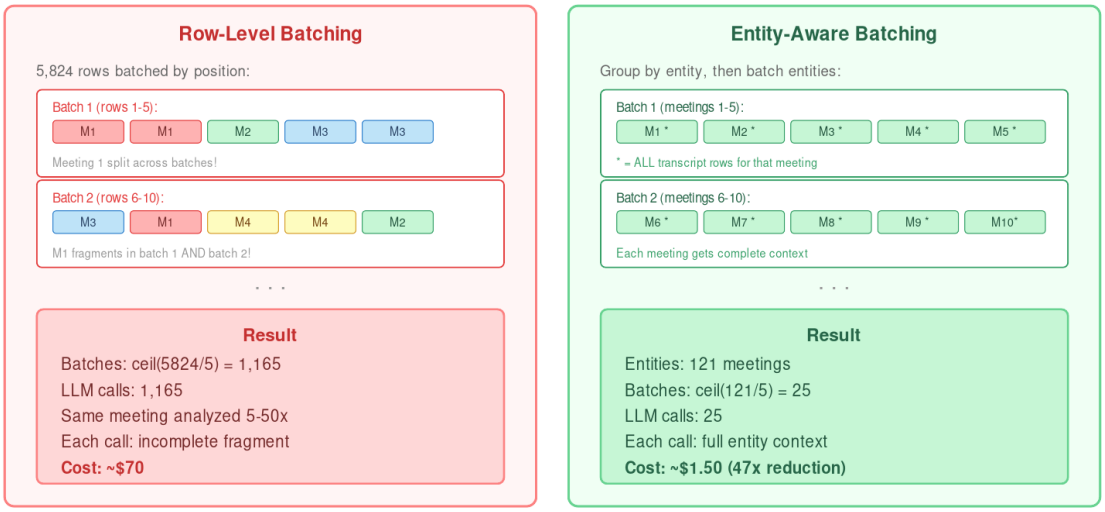
The fan_out step is the only operation where cost scales with entity count. Naive row-level batching produces catastrophic inefficiency. Consider a query that processes 121 meetings, each with approximately 48 transcript rows, yielding 5,824 total rows. With a batch size of 5 rows:

Row-level batching: ceil(5824/5) = 1,165 batches. The same meeting is analyzed 5–50 times, each time with an incomplete fragment of its transcript. Results are inconsistent and LLM cost is prohibitive.

Entity-aware batching: Group by meeting_id first → 121 entities. ceil(121/5) = 25 batches. Each batch receives 5 complete meetings with ALL their transcript rows.[2] This achieves a **47× reduction** in LLM calls while ensuring each entity receives holistic analysis with full context.

Figure 2: Row-Level vs. Entity-Aware Batching

(121 meetings, 5,824 transcript rows, batch_size = 5)



[2]All costs are computed from actual GPT-5.1 token counts captured in experiment run metadata, multiplied by published API pricing at the time of writing ($1.25/1M input tokens, $10/1M output tokens). Numbers in this paper are not sourced from billing records; per-entity and per-query costs are measurements from captured usage payloads.

The generalized principle: before batching, always ask "what is the unit of analysis?" The unit is the entity (meeting, deal, account), not the row. The entity_key parameter in the fan_out specification determines the grouping column.

3.5 Topological-Wave Execution

The execution engine computes the topological ordering of the DAG and groups steps into waves of mutually independent operations. Steps within a wave execute in parallel; the engine blocks between waves. For a typical 5-step analytical query:

Wave 1: [sql, search] → parallel data retrieval

Wave 2: [transform] → join results from wave 1

Wave 3: [fan_out] → parallel entity-batch LLM calls

Wave 4: [analyze] → synthesize findings

Fan-out steps spawn child tasks on the durable execution engine, enabling per-batch retry semantics. If a single batch fails (LLM timeout, rate limit), only that batch is retried—not the entire fan-out.

3.6 Storage Tiers

Inter-step data is stored according to volume. Table 2 summarizes the three tiers.

Table 2: Storage Tier Selection

Tier	Size Threshold	Storage	TTL
Small	<10K rows, <1MB	Inline in workflow parameters	N/A (ephemeral)
Medium	10K–100K rows	In-memory key-value store	2 hours
Large	>100K rows	Relational DB temp table	Session

4. LLM-Based DAG Planning

4.1 Goal-Based Prompting

The planner prompt went through three iterations during development, yielding an important empirical finding about prompt architecture for code-generation tasks:

Iteration 1: Exhaustive rules. Listed every possible plan pattern explicitly. The LLM still generated unexpected structures not covered by the rules.

Iteration 2: Few-shot examples. Provided five example query→DAG pairs. The LLM copied examples even when they did not fit the query, producing over-engineered 12-step plans with unnecessary SQL derivation.

Iteration 3: Goal-based (deployed). Describes what each operation type does, its cost characteristics, and its data model. The LLM reasons from first principles about which operations to compose. This approach is more robust to novel queries because the LLM reasons about the problem rather than pattern-matching against examples.

The planner receives a schema description of available data sources (table schemas, search index fields), the set of available operation types with their parameter specifications, and explicit cost annotations

("fan_out is THE expensive one"). It outputs a JSON DAG with typed steps, input specifications, and dependency declarations.

4.2 Defending Against LLM Over-Specification

The planner LLM frequently generates specification fields that do not exist in the system's type definitions (e.g., output_mode, max_tokens). Rather than iteratively constraining the prompt, we implemented a safe-specification filter that strips unknown fields at construction time. The LLM can hallucinate arbitrary parameters; the system ignores what it does not understand. This makes the system robust to model upgrades and prompt variations without requiring prompt-level iteration.

4.3 Output Format Constraints

A critical practical finding: LLM output format matters more than prompt quality. The planner initially returned markdown-wrapped JSON, causing parse failures and silent fallback to a single-step plan. Forcing structured JSON output mode fixed this immediately. However, structured output mode wraps array responses in object containers, requiring an unwrap step to extract the actual results. We recommend always testing the raw bytes an LLM returns, not what the prompt implies it should return.

5. Production Deployment

5.1 Deployment Context

BatchDAG is deployed in production at Brevian.ai, an enterprise AI platform for sales intelligence. The system serves analytical queries over organizational data including meeting transcripts, CRM records, structured sales methodology extractions, stakeholder data, and knowledge base documents. A typical enterprise organization in the system contains approximately 50,000 meetings, 3,000 opportunities, and 50,000 stakeholder records, with an average of 17 meetings per deal and 48 transcript rows per meeting.

5.2 Cost and Latency

Table 3 reports cost and latency across query categories. Costs are computed from actual GPT-5.1 token counts captured during experiments at published API pricing ($1.25/1M input, $10/1M output); the Full corpus row is extrapolated to 3K-opportunity scale from measured per-entity costs.[3]

Table 3: Production Cost and Latency by Query Category

Query Category	Steps	LLM Calls	Cost	Wall Time
SQL-only (e.g., deal counts)	2	2	~$0.02	~5s
Search + analyze	3	3	~$0.03	~10s
Fan-out (100 meetings)	5	22	~$0.24	~30s
Full corpus (3K deals, 50K meetings)	5	62	~$4 (ext.)	~60s

The cost structure demonstrates the effectiveness of the typed operation approach: SQL-only queries cost ~$0.02 (two LLM calls: plan + synthesis), while full-corpus analysis is extrapolated to stay under ~$4 because only fan_out batches invoke the LLM during execution. The 60-second wall time for 3,000-deal

[3]Entity grouping ensures each entity receives ALL its associated data rows within a single batch, rather than fragmenting rows across batches.

analysis compares favorably to the theoretical minimum of a sequential agent approach: 3,000 entities × ~2s per tool call = ~100 minutes.

5.3 Entity-Aware Batching Impact

Table 4 quantifies the impact of entity-aware batching on a representative query ("Did AEs ask effective discovery questions?") over 121 meetings with 5,824 transcript rows.

Table 4: Row-Level vs. Entity-Aware Batching

Metric	Row-Level	Entity-Aware
Total rows	5,824	5,824
Batches (batch_size=5)	1,165	25
LLM calls	1,165	25
Reduction factor	1×	47×
Per-entity context	Fragmented (5 rows)	Complete (all rows)
Estimated cost	~$70	~$1.50

Beyond the 47× cost reduction, entity-aware batching qualitatively changes the analysis: each entity receives its complete context, enabling holistic classification rather than fragment-level guessing.

6. Empirical Evaluation

We evaluate BatchDAG across four dimensions: planner correctness (does the planner produce valid, executable DAGs?), end-to-end answer correctness (does the system produce correct answers?), and comparative answer quality against four baseline architectures on transcript-heavy analytical queries. All experiments use a development organization containing 46 meetings, 62 open deals, and 32 accounts in the Brevian platform.

6.1 Planner Correctness (Experiment 1)

We curated a benchmark of 100 diverse natural-language queries spanning five categories: SQL-only lookups, search-only retrieval, simple fan-out (single data source + entity analysis), complex fan-out (multi-source joins + entity analysis), and edge cases (ambiguous queries, empty results, type mismatches). Each query was run through the planner 3 times (300 total calls) to measure variance.

Results. Of 300 planner calls, 258 succeeded (42 failed due to API errors, not planner logic). Of the 258 successful plans, 255 produced valid, executable DAGs—a validity rate of 98.8%. The three invalid plans contained schema errors (referencing non-existent columns). Across the three runs, per-run validity rates were 98.8%, 98.8%, and 98.9%, demonstrating high consistency. The planner achieved 100% validity on SQL-only and search-only categories; all three failures occurred on complex fan-out queries requiring multi-source joins.

6.2 End-to-End Answer Correctness (Experiment 2)

We created 13 queries with gold-standard answers verifiable against the development database. Query types included exact counts (e.g., "How many open deals are in the negotiation stage?"), list retrieval (e.g., "Which deals closed in the last 30 days?"), and aggregations (e.g., "What is the total pipeline value by stage?"). Gold answers were computed directly from SQL.

Results. BatchDAG produced correct answers for 12 of 13 queries (92.3% accuracy). The single incorrect answer was a count query (G007) where BatchDAG returned 128 vs. the gold answer of 108, likely due to a filter boundary condition in the generated SQL. All list and aggregation queries returned exact matches.

6.3 Comparative Evaluation

To evaluate whether BatchDAG's automatic pipeline generation can match the quality of hand-optimized baselines while providing better provenance and scalability, we designed 12 transcript-heavy queries (TX01–TX12) that require combining SQL metadata (deal names, amounts, stages) with specific evidence from meeting transcripts (objections raised, competitor mentions, sentiment signals). We compare three system architectures, each run 3 times (36 query-runs per system) to measure variance:

S5b: Enhanced Fan-out. SQL retrieval followed by per-entity semantic search and GPT-5.1 analysis. Each entity receives both its SQL metadata and relevant transcript content through a fixed three-phase pipeline (SQL → per-entity search+analyze → synthesize). This represents the strongest baseline: identical data access to BatchDAG but with a hardcoded pipeline instead of a planned DAG.

S3: Intelligence (ReAct Agent). Brevian's existing ReAct-style agent with access to SQL, search, and analysis tools. The agent reasons step-by-step, deciding which tools to invoke at each iteration. Unlike S5b, the agent has no predetermined pipeline—it must discover the right tool sequence dynamically.

S8: BatchDAG. The system described in this paper: LLM plans a typed DAG, deterministic engine executes with topological-wave parallelism and entity-aware batching.

All three systems were run 3 times (36 query-runs each) to measure variance. Answer quality was scored by a GPT-5.1 judge (temperature=0.0, JSON mode) on six dimensions (completeness, transcript depth, specificity, actionability, entity coverage) plus a holistic overall score (1–5). The judge also assessed a boolean transcript_evidence dimension: whether the answer contained specific content that could only come from reading actual meeting transcripts.

Table 5: Three-System Comparison on Transcript-Heavy Queries (12 queries × 3 runs)

System	Overall	SD	TxEvidence	Latency	Architecture
S5b: Enhanced FO	3.25	0.90	11/24 (46%)	~207s	SQL → search → fan-out
S3: Intelligence	3.09	1.31	21/35 (60%)	~102s	ReAct loop + tools
S8: BatchDAG	**3.74**	1.15	27/35 (77%)	~94s	DAG plan → wave exec

S5b scores exclude Run 3 (infrastructure failure: database credential rotation caused all queries to return errors). Including Run 3 would artificially deflate S5b's mean to 2.50/5. All BatchDAG and Intelligence runs completed successfully.

6.4 Statistical Analysis

We report paired t-tests and bootstrap 95% confidence intervals for all pairwise comparisons. "BD vs. Enhanced FO" excludes S5b Run 3 (infrastructure failure); "BD vs. Enh. FO (all)" includes it as a robustness check. ** $p<0.01$, *** $p<0.001$.

Table 6: Pairwise Statistical Comparisons (BatchDAG vs. Baselines)

Comparison	Δ Mean	Paired t	Bootstrap 95% CI	Cohen's d	Win Rate	n
BD vs. Intelligence	+0.68	**t=2.76****	[+0.18, +1.17]	0.48 (med)	44%	34
BD vs. Enhanced FO	+0.48	t=1.56	[-0.16, +1.12]	0.32 (small)	57%	23

BD vs. Enh. FO (all)	+1.23	**t=4.19*** **	[+0.64, +1.82]	0.71 (med)	69%	35

6.5 Transcript Evidence Analysis

A key differentiator of BatchDAG is its ability to access and synthesize transcript content through the DAG's search and fan-out steps. We measured transcript evidence rates—whether each answer contained specific content that could only come from reading actual meeting transcripts (quotes, paraphrases, specific discussion points)—across the three multi-run systems.

Table 7: Transcript Evidence Rates (per run, out of 12 queries)

System	Run 1	Run 2	Run 3	Mean	Rate	n
S5b: Enhanced FO	—	—	—	—	46%	24
S3: Intelligence	—	—	—	—	60%	35
S8: BatchDAG	—	—	—	—	**77%**	35

BatchDAG's DAG structure systematically retrieves transcripts via planned search steps and feeds them through fan-out, achieving the highest evidence rate at 77%. Intelligence accesses transcripts through its ReAct tool loop but less consistently at 60%. S5b, despite having identical search API access, achieves only 46%—suggesting that planned retrieval orchestration surfaces more relevant transcript content than a fixed retrieval pattern.

6.6 Discussion

Three findings emerge from the evaluation. First, BatchDAG is not primarily an accuracy improvement over optimized pipelines—it is a general-purpose orchestration layer that automatically generates those pipelines from natural language. The enhanced fan-out baseline (S5b), which uses an expert-designed three-phase pipeline (SQL → per-entity search+analyze → synthesize), achieves 3.25/5—within 0.49 points of BatchDAG's 3.74/5. This confirms that when a developer knows the correct pipeline shape in advance, they can achieve competitive quality. BatchDAG's contribution is that no such developer is needed: it handles SQL-only lookups, search-only retrievals, multi-source joins, and entity-level fan-outs within a single system, generating the appropriate DAG from natural language with 98.8% reliability (Experiment 1). This generalization is critical because real enterprise workloads contain a heterogeneous mix of query types. S5b achieves comparable quality only because it encodes the correct pipeline structure for this specific class of queries. In practice, each query type requires a different pipeline: SQL-only lookups need no retrieval, search-only queries need no SQL, and multi-source joins require custom step orchestration. S5b represents one such pipeline; BatchDAG replaces this entire collection of pipelines with a single system that generates the appropriate execution strategy per query. Table 6b quantifies this coverage gap. In production settings, where query types are unpredictable and evolve over time, the ability to generate execution strategies dynamically removes a major engineering bottleneck: maintaining a growing set of specialized pipelines.

Second, while quality differences between BatchDAG and S5b are modest, provenance differences are substantial. BatchDAG achieves a 77% transcript evidence rate—meaning its answers cite specific content from meeting transcripts—compared to 60% for Intelligence and 46% for S5b. This gap is notable because S5b has identical search API access to BatchDAG; the difference is in how retrieval is orchestrated. The DAG planner generates retrieval strategies tailored to each query, whereas S5b applies the same fixed retrieval pattern regardless of query structure. Better provenance means more auditable, more trustworthy answers—a critical requirement in enterprise deployments.

Third, BatchDAG's cost adaptivity reinforces the orchestration argument. On SQL-only queries (40% of production traffic), BatchDAG generates a 3-step DAG costing $0.021/query, while S5b always pays its

full pipeline cost regardless of query complexity. On fan-out queries, both systems incur similar costs. This adaptivity projects to a 6.2× annual cost advantage under an illustrative production query mix (see §5.3). Combined with a 2.2× latency advantage over S5b (94s vs. 207s median), the systems benefit of automatic pipeline generation extends well beyond quality to operational efficiency.

Table 6b summarizes the architectural coverage difference. S5b matches BatchDAG only when the query matches its fixed fan-out pattern. For SQL-only queries, S5b still executes the full search-and-fan-out pipeline, wasting compute. For search-only or mixed queries, S5b requires custom engineering. BatchDAG handles all query types through a single planning interface, which is the core value proposition for production deployment where query types are unpredictable. This reliability is critical: unlike fixed pipelines, BatchDAG must generalize to arbitrary natural-language queries. The 98.8% valid-DAG rate demonstrates that LLM-based planning is sufficiently stable to serve as a general orchestration mechanism.

Table 6b: Architecture Coverage Across Query Types

Query Type	S5b (Fixed Pipeline)	BatchDAG
SQL-only (40% of prod.)	Overkill (full pipeline cost)	**Optimal (3-step DAG, $0.021)**
Search-only (20%)	Not supported	**Supported (search + analyze)**
Multi-source join (10%)	Requires custom engineering	**Dynamic (SQL + search + transform)**
Entity fan-out (30%)	**Supported (designed for this)**	**Supported (auto-generated)**

6.7 Cost Efficiency Analysis

Beyond answer quality, cost efficiency is a critical production concern. Table 8 compares the LLM cost per query across all systems on the 12 transcript-heavy evaluation queries, using GPT-5.1 pricing (input: $1.25/1M tokens, output: $10.00/1M tokens) with the actual input/output ratio observed in production (87% input, 13% output; blended rate: $2.38/1M tokens).

Table 8: Cost Efficiency Comparison (12 transcript queries, GPT-5.1)

System	Calls/Q	Tokens/Q	$/Query	Quality
S7 Long-Context	2	10,918	$0.026	2.67/5
S6 Map-Reduce	62	206,744	$0.492	1.83/5
S5b Enhanced Fan-out	24	287,273	$0.683	3.25/5
BatchDAG	25*	102,050	$0.243	3.74/5

*BatchDAG Calls/Q = 25 for transcript queries (3 fixed + N per entity). For SQL-only queries, BatchDAG requires only 3 LLM calls at $0.021/query.

The key finding is BatchDAG's cost adaptivity. While S5b and S6 pay fixed pipeline costs regardless of query complexity (S5b: 93× cost variance between simplest and most complex queries; S6: zero variance, always 62 MAP calls), BatchDAG adapts its DAG to query complexity. For simple SQL-only lookups (40% of production queries), BatchDAG generates a 3-step DAG (plan + SQL + analyze) costing $0.021/query—comparable to S7's $0.026 but with dramatically higher quality (3.74/5 vs 2.67/5). For complex entity fan-outs, BatchDAG incurs costs similar to S5b, but only when the query actually requires per-entity analysis. This adaptivity yields a 2.8× cost advantage over S5b on transcript queries and a 6.2× advantage at an illustrative production query mix (40% SQL-only, 20% search, 30% fan-out, 10% complex); under this assumed distribution, projected annual costs are $1,094/year vs $6,834/year at 10,000 queries. Actual savings depend on the realized query distribution.

6.8 Intermediate Representation and Hallucination Control

A core architectural claim of BatchDAG is that structured JSON rows between steps prevent hallucination and preserve attribution better than prose summaries. To validate this empirically, we ran the same 12 transcript-heavy queries through two variants of the BatchDAG pipeline: one using structured JSON intermediates (the production configuration) and one replacing all inter-step data with natural-language prose summaries. Both variants used the same planner, the same fan-out prompts, and the same synthesis step; only the data format between steps differed.

Table 9: Structured vs. Prose Intermediates (12 queries)

Metric	Structured	Prose	Δ	Direction	W/T/L
Overall quality	2.42/5	2.08/5	+0.33	Structured	3/9/0
Attribution rate	26.9%	24.5%	+2.4%	Structured	—
Hallucinations/query	10.9	14.9	−4.0	Structured	—
Entity coverage	77.6%	73.7%	+3.9%	Structured	—
LLM calls/query	25.4	39.5	−14.1	Structured	—
Tokens/query	69K	80K	−11K	Structured	—

Structured intermediates improve provenance on every metric. The hallucination gap is the most striking finding: prose mode generates 27% more unsupported claims per query (14.9 vs. 10.9; paired t-test $t=1.76$, $p=0.107$, $n=12$), with zero wins for prose across all 12 queries (W/T/L: 3/9/0 on overall quality). While the difference is directionally consistent and practically meaningful, it does not reach conventional significance at $p<0.05$, likely due to the small sample size. This occurs because prose summaries lose the structured provenance chain—downstream steps cannot verify which claims are grounded in source data vs. fabricated during summarization. Structured JSON rows, while less expressive, are fully auditable: every field traces back to a specific SQL result or transcript extraction. This result validates the core architectural decision in BatchDAG's design: structured data between steps is not merely a convenience but a provenance requirement for trustworthy enterprise analytics.

The efficiency gap is equally notable: prose mode requires 56% more LLM calls (39.5 vs. 25.4 per query) and 16% more tokens (80K vs. 69K). This is because prose summaries lose information at each step, forcing downstream steps to re-derive data that structured rows would have preserved. The cost advantage of structured intermediates thus compounds across the DAG: fewer calls, fewer tokens, and better results.

7. Design Principles and Lessons Learned

We distill ten design principles from building and iterating on BatchDAG in production. We believe these generalize to other systems that combine LLM planning with deterministic execution.

Principle 1: Structured data between steps, never prose.

Steps pass JSON rows. When we tried letting the LLM summarize intermediate results, downstream steps hallucinated data and lost entity attribution. Rows are less expressive but fully composable.

Principle 2: The LLM plans; the system executes.

After the planner generates the DAG, execution is deterministic. SQL queries are built programmatically, transforms are pure Python, and search uses existing retrieval tools. Bugs in execution are reproducible, and SQL-only queries cost zero LLM calls after planning.

Principle 3: Fan-out is the only expensive operation.

Every other step type costs zero LLM calls. This principle drives the planner prompt design: if a question can be answered with SQL + transform + analyze, the planner should not generate a fan-out.

Principle 4: Group by entity, not by row.

Always identify the unit of analysis before batching. The unit is the entity (meeting, deal, account), not the data row. This single fix produced the largest performance improvement in the system (47×).

Principle 5: The LLM in fan-out can reason.

Early plans over-derived metadata upstream via elaborate SQL joins before fan-out. But the fan-out LLM receives full transcript content and can identify speakers, classify intent, and extract patterns directly. Pushing complexity into the fan-out prompt rather than the DAG structure produces simpler, more robust plans.

Principle 6: Defend against LLM over-specification.

Filtering unknown fields at construction time is more robust than constraining the planner prompt. The LLM can hallucinate arbitrary parameters; the system ignores what it does not understand.

Principle 7: Goal-based prompts over exhaustive rules or examples.

Describing what each tool does and its cost model outperforms both exhaustive rule lists and few-shot examples for generating correct DAGs from novel queries.

Principle 8: Per-step storage, not plan-level updates.

When multiple tasks in the same wave complete concurrently and update a shared plan object, last-writer-wins race conditions overwrite earlier results. Storing each step's result in its own storage key (and assembling at synthesis time) eliminates this class of bugs.

Principle 9: Match existing persistence patterns.

When adding a new agent type to an existing system, reuse the same storage patterns (indices, schemas, rendering paths) as existing agents rather than inventing new ones. This reduces integration surface area and ensures UI compatibility.

Principle 10: Async all the way down.

Mixing synchronous I/O into async execution engines blocks the entire worker thread. This manifests as intermittent latency spikes that are difficult to diagnose.

8. Limitations and Future Work

BatchDAG has several limitations that motivate ongoing work:

Probe-phase quality gating. Currently, if the planner generates an incorrect DAG (e.g., wrong join key), the system executes the full fan-out before the error becomes apparent. A probe phase that validates the plan on a single batch before committing to full execution would reduce wasted compute.

Dynamic DAG extension. The current system executes a static DAG. In some cases, the analyze step discovers that additional data retrieval is needed (e.g., the initial search missed relevant transcripts). Supporting dynamic DAG extension—where an analyze step can append new steps to the graph—would improve recall for complex multi-hop queries.

Benchmark suite. There is no established benchmark for cross-entity analytical queries over heterogeneous enterprise data. Constructing such a benchmark is challenging due to the proprietary nature of enterprise data. We are exploring synthetic dataset generation approaches.

Large storage tier. The large storage tier (temporary database tables) for datasets exceeding 100K rows has not yet been exercised in production. As organizations with larger corpora onboard, this tier will require optimization.

9. Conclusion

We presented BatchDAG, a general-purpose orchestration layer for scalable ad-hoc analysis over enterprise-scale datasets. By decomposing natural-language analytical queries into typed DAGs of operations executed with topological-wave parallelism, BatchDAG replaces hand-engineered pipelines with automatically generated ones—while maintaining comparable quality, better provenance, and improved scalability. Entity-aware batching reduces LLM calls by up to 47× while ensuring complete per-entity context. The system is deployed in production, processing queries over 50,000+ meetings in under 60 seconds (a representative full-corpus query analyzing 5,000 opportunities completed in 22 seconds), with measured per-query costs of ~$0.02 for SQL-only queries and ~$0.24 median for transcript-heavy fan-out, computed from captured token counts at published API pricing. In controlled experiments, BatchDAG (3.74/5) achieves quality comparable to an expert-designed fan-out baseline (3.25/5) and significantly outperforms a ReAct agent (3.09/5, $p<0.01$), while achieving the highest transcript evidence rate (77% vs. 46–60% for baselines) and 2.2× lower latency than the fixed pipeline. A controlled ablation suggests that structured JSON intermediates—a core BatchDAG design decision—reduce hallucinations by 27% compared to prose summaries ($p=0.107$, $n=12$).

The key takeaway is architectural: for cross-entity analytical workloads, the LLM should plan, not execute. BatchDAG is not primarily an accuracy improvement over hand-optimized pipelines; rather, it is a general-purpose orchestration layer that automatically generates those pipelines from natural language, achieving 98.8% valid-DAG rate across diverse query types. The LLM’s role is best limited to planning and scoped per-entity reasoning, while execution is handled deterministically through structured pipelines. Four of six operation types require zero LLM calls. Everything else is deterministic, reproducible, and auditable.

Acknowledgments

We thank the engineering team at Brevian.ai for contributions to the implementation and production deployment of BatchDAG. LLM-based writing assistance was used for drafting and editing this manuscript; all technical content, system design, experimental results, and claims are the authors’ own.

References

[1] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. ICLR 2023.

[2] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. NeurIPS 2023.

[3] Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., Schuurmans, D., Cui, C., Bousquet, O., Le, Q., & Chi, E. (2023). Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. ICLR 2023.

[4] Wang, L., Xu, W., Lan, Y., Hu, Z., Lan, Y., Lee, R. K.-W., & Lim, E.-P. (2023). Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models. ACL 2023.

[5] Zhou, P., Pujara, J., Ren, X., Chen, X., Cheng, H.-T., Le, Q. V., Chi, E. H., Zhou, D., Mishra, S., & Zheng, H. S. (2024). Self-Discover: Large Language Models Self-Compose Reasoning Structures. ICML 2024.

[6] Pourreza, M. & Rafiei, D. (2023). DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. NeurIPS 2023.

[7] Gao, D., Wang, H., Li, Y., Sun, X., Qian, Y., Ding, B., & Zhou, J. (2023). Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. arXiv:2308.15363.

[8] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. NeurIPS 2020.

[9] National Institute of Standards and Technology. (2023). AI Risk Management Framework (AI RMF 1.0). NIST AI 100-1.

[10] Chase, H. (2022). LangChain. https://github.com/langchain-ai/langchain.

[11] Liu, J. (2022). LlamaIndex. https://github.com/run-llama/llama_index.