



Multi-Objective Agentic Rewrites for Unstructured Data Processing

Lindsey Linxi Wei^{1†}, Shreya Shankar^{2†}, Sepanta Zeighami²,
Yeounoh Chung³, Fatma Ozcan³, Aditya G. Parameswaran²

¹University of Washington, ²UC Berkeley, ³Systems Research @ Google

linxiwei@cs.washington.edu, {shreyashankar, zeighami, adityagp}@berkeley.edu, {yeounoh, fozcan}@google.com

ABSTRACT

One year ago, we open-sourced [DocETL](#), a declarative system for LLM-powered data processing that, as of March 2026, has 3.7K GitHub stars and users across journalism, law, medicine, policy, finance, and urban planning. In DocETL, users compose operators described in natural language, with an LLM executing each operator’s logic. However, due to complexity in operators or the data, LLMs often give inaccurate results. DocETL addressed this with *rewrite directives*: abstract rules that guide LLM agents in rewriting pipelines by decomposing operators or data—for example, splitting a single `filter("is this email sent from an executive and discussing fraud?")` into two separate semantic filters. Yet DocETL optimizes only for accuracy, not cost. *How do we optimize for both?*

We present **MOAR** (Multi-Objective Agentic Rewrites), a new optimizer for DocETL. To optimize cost, we add two new categories of directives and extend all three existing ones, bringing the total to over 30—more than doubling DocETL’s original set of directives. Moreover, because operators interact unpredictably due to LLM behavior, optimizing them in isolation yields suboptimal plans. So, we design a new global search algorithm that explores rewrites in the context of entire pipelines. Since this space is infinite—every pipeline can be rewritten, and each rewrite rewritten again—we adapt a multi-armed bandit framework to prioritize which pipelines to rewrite. Across six workloads, MOAR achieves **27% higher accuracy** than ABACUS, the next-best optimizer, while matching its best accuracy at **55% of its cost**.

PVLDB Reference Format:

Lindsey Wei, Shreya Shankar, Sepanta Zeighami, Yeounoh Chung, Fatma Ozcan, Aditya G. Parameswaran. Multi-Objective Agentic Rewrites for Unstructured Data Processing. PVLDB, 19(11): 3062 - 3075, 2026.
doi:10.14778/3836663.3836673

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at github.com/ucbecpic/docetl/blob/main/experiments/reasoning/RUNMOAR.md.

1 INTRODUCTION

LLMs are now integrated into systems that support queries over unstructured data, from both industry and academia [18, 21, 30, 36, 44, 48, 57]. In these LLM-powered data processing systems, a

query is expressed as a combination of *semantic operators*. Semantic operators are data processing operators such as `map`, `reduce`, and `filter`, each described in natural language for an LLM to carry out. Users define an initial pipeline, and the system’s optimizer then determines how to execute it. DocETL [44] is one such system. Consider the following example of a DocETL workload from the public defender’s office in a major California city:

Example 1.1 (Public defender workloads). Our public defender users represent defendants whose sentences were increased due to *enhancement factors*—circumstances like threatening with a firearm, causing severe injury, or kidnapping. To investigate whether enhancement factors are applied equitably across racial groups, defenders want to extract evidence of factors from tens of thousands of pages of police reports and trial transcripts, then compare which factors were actually charged by the court. The pipeline is typically a single operation: `map("given a description of these [eight] types of enhancement factors...list each factor present, along with supporting evidence")`.

Query optimization in settings like the one above is challenging. Unlike traditional data processing, where the optimizer only minimizes cost, the query plan must also be *accurate*. If the accuracy is too low (e.g., below 95% precision as per guidelines from public defenders), the plan’s output is not useful. The optimizer should surface high-accuracy plans that span a range of costs, so users can select the one best aligned with their budget.

Limitations of Existing Systems. Query optimizers in data systems search over rewrites (i.e., transformations that restructure queries) to find efficient plans. Search typically relies on the principle of *optimal substructure*: an optimal plan can be constructed from optimal solutions to its subplans.¹ For example, the Cascades framework [16] organizes subexpressions into equivalence groups, and the optimal plan for a subexpression is reused wherever it appears. However, in LLM-powered data processing, even when optimizing only for accuracy, the same subplan can produce outputs of varying benefit, *depending on the subplans that precede or follow it*. For instance, in Ex. 1.1, suppose we decompose the pipeline into a `map` per factor type, followed by a `reduce` that unifies the evidence. A `map` that extracts smaller or fewer text spans may achieve higher accuracy in isolation, but a `map` that includes more surrounding context may yield better overall accuracy if the downstream `reduce` can leverage that context to deduplicate extractions or filter out false positives. Which `map` implementation is optimal depends on which `reduce` implementation that follows—so we cannot optimize them independently. **More generally, since individual subplan choices interact in unpredictable ways, composing locally optimal subplans can yield globally suboptimal plans.**

¹Even when physical properties like sort order make subplan costs context-dependent, we can restore optimal substructure, by treating each (expression, property) pair as a distinct subproblem.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836673

[†]Co-first authors. Lindsey was a visiting research intern at UC Berkeley.

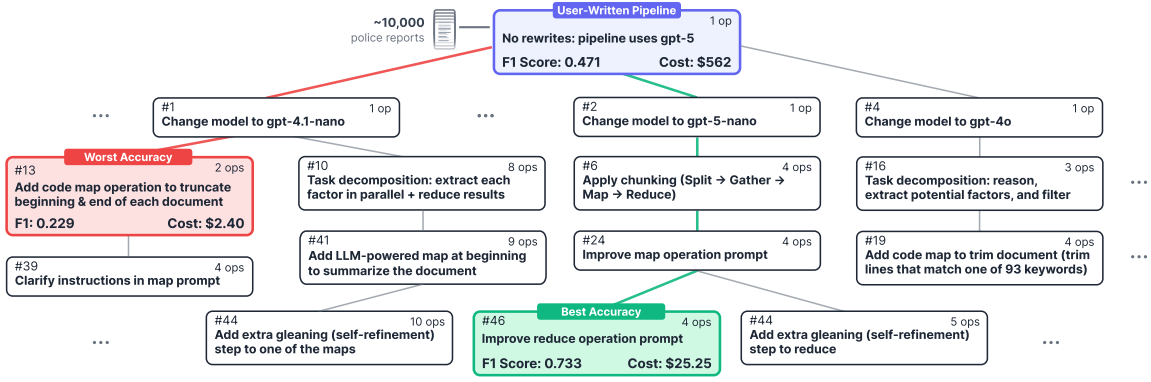


Figure 1: Sample of pipelines explored by MOAR when optimizing the pipeline in Ex. 1.1. The user-written pipeline (top, purple) contains a single map operator that extracts all enhancement factors. Each node is a pipeline variant produced by a rewrite (e.g., model change, code synthesis, task decomposition, data decomposition). The “...” symbols denote other explored pipelines. The best pipeline (#46, green) achieves the highest F1 score while also costing less than the user’s original plan.

Existing optimizers for LLM-powered data processing have limitations. LOTUS [36] only minimizes cost, requiring users to author an accurate plan first, which is difficult for users. LOTUS ignores the pipeline search problem entirely by considering only one optimized implementation per operator. ABACUS [41] performs cost-based search using rules from traditional data processing (e.g., filter pushdown) and new implementation rules (i.e., logical-to-physical) for semantic operators (e.g., model selection, prompting strategies, ensembling). ABACUS intelligently samples operator implementations to estimate their cost and accuracy, then uses the Cascades search framework [16]—which relies on optimal substructure—to find Pareto-optimal query plans. Our original DocETL optimizer [44] (that we call DocETL-V1) relaxes the optimal substructure assumption slightly, but still optimizes operators from upstream to downstream, thus missing rewrites where the best upstream choice depends on downstream operators. Moreover, DocETL-V1 optimizes only for accuracy, not cost.

The MOAR optimizer. In this paper, we present MOAR, an optimizer that automatically discovers pipeline variants that improve accuracy over the user’s original while offering a range of costs. MOAR explores rewrites and returns a Pareto frontier of options spanning different accuracy-cost trade-offs, all under a limited evaluation budget. MOAR consists of two key components: (1) a library of rewrite directives that define transformations over pipelines, and (2) a novel search algorithm that does not assume optimal substructure. For (1), following DocETL, we use *rewrite directives*—abstract rules that are instantiated by LLM agents into concrete pipeline rewrites based on task semantics and sample data. We substantially extend DocETL’s directive library to target both accuracy and cost. For (2), we model the space of pipelines (i.e., query plans) as a graph where each node is a complete pipeline, and each edge applies a rewrite to produce a new one. MOAR iteratively (i) selects a promising pipeline, (ii) applies a rewrite to produce a variant, and (iii) evaluates it on samples to estimate accuracy and cost.

New Rewrite Directives. The first challenge in designing MOAR was creating rewrite directives that reduce cost while maintaining or improving accuracy. We introduce two new categories of directives: *code synthesis*, which replaces LLM-powered operators with synthesized Python code, and *operator fusion and reordering*, which combines multiple operators into fewer ones or reorders

them—with 4 and 5 directives in each. For operator fusion, same-type operators can obviously be fused (e.g., two maps combined by unioning output schemas and merging prompts). But different-type operators can be fused too: a map followed by a filter can become a single map whose prompt incorporates the filter logic and generates an additional Boolean attribute, followed by a non-LLM based predicate that drops documents based on that attribute. We also extend DocETL’s three existing categories with new cost-reducing directives. For example, we extend projection synthesis (a category of directives that insert map operations prior to more complex tasks) with directives that insert maps to *compress* documents, reducing the amount of text that downstream semantic operators handle. MOAR adds 18 deployment-informed directives to DocETL, bringing the total to over 30.

New Search Algorithm. Another challenge is designing an *efficient* global search algorithm. Since evaluating a pipeline requires executing it on sample data, we can only explore a limited number of pipelines. Rewrites can be applied in sequence, creating a vast search space, as shown in Fig. 1. For instance, the highest-accuracy pipeline (#46, green) is discovered through four rewrites: first switching to a cheaper model (#2), then applying data decomposition to process document chunks (#6), followed by prompt improvements to the map (#24) and reduce (#46) operations. While LLM agents can determine which directives to apply and how to instantiate them, it is unclear which pipelines will lead to high-accuracy, low-cost descendants. **Our insight is to learn which pipelines are promising to rewrite, by adapting multi-armed bandits.** We use a variant of the UCB (Upper Confidence Bound) algorithm, adapted for the tree setting [6], to select which pipeline to rewrite. However, we define a custom metric rather than using the typical hypervolume metric from multi-objective optimization [60], which treats all Pareto frontier points as equally valuable. In LLM-powered data processing, low-accuracy pipelines vastly outnumber high-accuracy ones, so optimizing for hypervolume would waste the evaluation budget exploring low-accuracy regions of the search space. We instead introduce a metric based on *marginal accuracy contribution*—the vertical distance between a pipeline’s accuracy and the best accuracy at comparable cost (shown as the red line in Fig. 2)—and score each pipeline by aggregating this metric across the pipeline and its descendants.

Then, with over 30 directives, each applicable to multiple operators, a single pipeline could spawn hundreds of children, exhausting the evaluation budget on variants of one pipeline rather than exploring deeper rewrite sequences. We employ *progressive widening* [10], a technique that limits how many edges (i.e., immediate rewrites) a node (i.e., pipeline) can have based on its visit count (i.e., total number of descendants of any depth). As a node accumulates more visits, it is allowed to have more edges, but its edge growth is sublinear, forcing the search to explore other regions of the graph before returning to generate additional variants from any single pipeline.

Overall, the contributions of this paper include:

- **An expanded and extensible library of rewrite directives (Sec. 3).** We design and implement an extensible library of 18 new directives, greatly improving the expressive power of DocETL’s original 13 directives. These include directives targeted at cost reduction (e.g., model substitution, context truncation, operator fusion), as well as directives that replace LLM calls with synthesized code implementations.
- **A search algorithm for multi-objective optimization (Sec. 4).** We introduce a new global search algorithm for discovering sequences of rewrites that improve both accuracy and cost while operating under a limited number of pipeline evaluations.
- **Empirical evaluation across six workloads (Sec. 5).** We evaluate MOAR on six real-world workloads spanning legal, medical, and enterprise domains. Compared to state-of-the-art systems and naive agentic baselines, **MOAR achieves up to 11× higher accuracy and up to 99% lower inference cost** at equivalent accuracy levels, while dominating the Pareto frontier of cost and accuracy **on all workloads**. On average, compared to the next best optimizer (ABACUS [41]), MOAR achieves 27% higher accuracy, while matching its best accuracy at only 55% of its cost.

MOAR is open-source: documentation is available at [this link](#).

2 BACKGROUND AND DEFINITIONS

We build our optimizer on top of the DocETL [44] system (i.e., DSL, parser, and execution engine), though our techniques can extend to other systems. We write DocETL-V1 when referring to contributions from Shankar et al. [44], such as the original query optimizer and rewrite directives. Secs. 2.1 and 2.2 describe background on DocETL and semantic operators. Sec. 2.3 describes the optimization problem setup. Table 2 summarizes all notation.

2.1 Datasets, Operators, and Pipelines

Datasets. A *dataset* D is a collection of documents. Users can provide datasets in JSON or CSV formats, where each document is a set of key–value pairs, each representing metadata or free-form text. In the workload from Ex. 1.1, each document has a `case_id` field and a `notes` field containing the report text, which can be tens or hundreds of pages.

Semantic Operators. We refer to operators that transform data using natural language (NL) specifications as *semantic operators*, following Patel et al. [36]. Each semantic operator in DocETL has four components: (i) an *operator type* such as `map`, `filter`, or `reduce` (e.g., `map` applies a transformation to each document, `reduce` aggregates groups of documents into one); (ii) a *prompt template* written in natural language that describes the operator’s semantics, expressed in Jinja [40]; (iii) an *output schema* that declares the structure of

the operator’s results; and (iv) a *model* specifying the LLM used to execute the operator (e.g., `gpt-4o-mini`). We denote M as the set of models that the user makes available to the optimizer, such as GPT or Gemini variants, and by $m \in M$ a particular model. While the user selects a specific m when authoring an operator, the optimizer is free to choose any model from M for any semantic operator, or replace LLM execution with *code-powered* implementations—synthesized code that realizes the task specified by the operator’s prompt template and output schema.

Formally, we denote a semantic operator generically by o . When we need to make its configuration explicit, we write o_x , where $x = (p, s, m)$ denotes its prompt template p , output schema s , and model $m \in M$. Applying o_x to a dataset D produces a new dataset $D' = o_x(D)$, with a schema following s . When the operator type is important, we write map_x , reduce_x , and so on. Unless otherwise noted, o_x (and typed forms like map_x , reduce_x) denote an LLM-powered semantic operator. For code-powered implementations, m is set as $\emptyset$ and p contains synthesized Python code that obeys the NL specification, with output obeying schema s . We may write `code_mapx`, `code_reducex`, etc., to emphasize the fact that an operator has a code-powered implementation. Relational operators are a special case of code-powered operators.

Consider the following example pipeline in Ex. 1.1. Let map_x have configuration $x = (p, s, m)$, where p is the prompt template: “Given the text in `{{ input.notes }}`, return all the enhancement factors present, along with supporting evidence.” s is the output schema: `enhancements: list[{{factor: str, evidence: str}}]`, and m is the model (e.g., `gpt-4o-mini`). For a dataset D , the result $\text{map}_x(D)$ is a dataset where each document now includes a new key (or attribute) `enhancements`, derived using its notes.

Pipelines. Given operators, either LLM or code-powered, a *pipeline* P is then a sequence of k semantic operators $(o^{(1)}, \dots, o^{(k)})$; given an input dataset D , we write its execution as $o^{(1)} \rightarrow o^{(2)} \rightarrow \dots \rightarrow o^{(k)}$, where $\rightarrow$ denotes function composition, following Shankar et al. [44]: $o^{(k)}(o^{(k-1)}(\dots o^{(1)}(D) \dots))$. While users specify operators via NL prompt templates, schemas, and model choices, these serve as a baseline specification. The optimizer may rewrite pipelines by substituting models or replacing LLM-powered implementations with code-powered ones (including relational operators), decomposing operators into multiple ones, or fusing multiple operators, to discover cheaper or more accurate alternatives. DocETL-V1 supports six semantic operator types (`map`, `parallel_map`, `reduce`, `filter`, `resolve`, `equijoin`) and three non-LLM operators (`split`, `gather`, `unnest`). MOAR extends this with additional operator types detailed in Sec. 3. In DocETL, pipelines and operators are specified in YAML [7]: a pipeline is a list of operator configurations, where each operator’s *configuration* is represented as a dictionary of parameters (such as prompt template, output schema, and model).²

2.2 Rewrites and Directives

A *rewrite* transforms a pipeline P into a new pipeline P' . Formally, if $P = (o_1, \dots, o_k)$, a rewrite r replaces a subsequence $(o_i, o_{i+1}, \dots, o_j)$ with $(o'_1, o'_2, \dots, o'_\ell)$, yielding a new pipeline P' . Intuitively, rewrites modify one or more operators in P to produce an alternative pipeline.

²For simplicity, we define pipelines as linear sequences, which cover the predominant use cases in our deployments and all workloads in our evaluation, but our subsequent rewriting techniques generalize to tree-structured pipelines.

A *rewrite directive* is a transformation rule that induces rewrites, analogous to rewrite rules in traditional databases. A directive d consists of: (i) a *left-hand side (LHS)*, i.e., a pattern over operator types (and optional conditions on their configurations) that must match a subsequence of operators in the pipeline, and (ii) a *right-hand side (RHS)* that specifies the new operator sequence to substitute, and how their configurations (prompt templates, schemas, models, or code) are to be constructed. As in DocETL-V1, we call them *rewrite directives* instead of rewrite rules because, unlike traditional rules that are fully specified, directives are abstract patterns requiring concrete instantiation of operator configurations. LLM agents instantiate these directives.

To apply a directive d , the optimizer selects a target subsequence that matches d 's LHS, then instantiates d by generating concrete configurations for the RHS operators, yielding a specific rewrite r . d is the general rule, and r is one concrete instantiation. For example, one directive from DocETL-V1 has LHS $\text{map}_{x'}$ and RHS $\text{code_split} \rightarrow \text{code_gather} \rightarrow \text{map}_{x'} \rightarrow \text{reduce}$, where x' is a modified version of x adapted to process chunks rather than full text. This directive splits the largest text field in each document into chunks (we colloquially refer to this text field as the “document” for simplicity; when we mean the full JSON object later, we will say “document JSON object”), augments each chunk with “peripheral” context from elsewhere in the document, applies the map to augmented chunks, and aggregates results—improving accuracy when text is too long for the LLM’s context window. The *reduce* operator (subscripts omitted) is newly synthesized to aggregate chunk-level results. For this directive, DocETL-V1 generates *code_split* and *code_gather* configurations non-agentically by trying different chunk sizes or peripheral contexts, and uses an LLM agent to generate the prompt template and schema for $\text{map}_{x'}$ and the new *reduce* operator.

2.3 Optimization Problem Setup

We now formalize our optimization problem. We focus on monetary cost as our primary cost metric, though our framework can be extended to capture other costs such as latency.

Cost Model. Each operator has an associated *cost*. For an LLM-powered operator o_x with configuration $x = (p, s, m)$, the cost $c(o_x)$ is typically proportional to the number of input and output tokens in p and s , multiplied by the per-token price of model m , and by the number of JSON documents o_x processes. For a code-powered operator *code_op*, the cost is set to $c(\text{code_op}) = 0$, since we want to minimize monetary cost. The cost of a pipeline $P = (o_1, \dots, o_k)$ is the sum of individual operator costs: $c(P) = \sum_{i=1}^k c(o_i)$. Each pipeline is evaluated by a user-defined scoring function $a(\cdot)$ applied to its final output on dataset D , i.e., $a(P) = a(P(D))$. The function a may include accuracy, precision, recall, or other application-specific metrics; if a requires ground truth labels, the user provides them, or a can be an LLM-as-judge implementation [59]. For simplicity, we refer to this objective as “accuracy,” with $a(P) \in [0, 1]$. Our goal, therefore, is to surface high-accuracy pipelines that offer trade-offs between accuracy and cost. We formalize this goal using the notion of Pareto optimality.

Definition 2.1 (Pareto set). For any set of pipelines S , $\text{Pareto}(S) = \{P \in S : \{P' \in S \mid a(P') > a(P), c(P') \leq c(P), P' \neq P\} = \emptyset\}$.

Thus the Pareto set of S is the set of pipelines that are not dominated (on both cost and accuracy) by any other member of S .

Objective. Let $\mathcal{P}$ denote the set of pipelines reachable from the initial user-authored pipeline P_0 through any sequence of rewrites. $\mathcal{P}$ is clearly infinite, as there are an arbitrary number of ways to instantiate a given rewrite directive. Let B denote the evaluation budget, i.e., the maximum number of candidate pipelines that can be executed and scored. We define the budget in terms of pipeline evaluations because, in our experience, users can reason about “how many pipeline variants to explore” more easily than a monetary budget, which depends on unfamiliar factors like LLM pricing and data characteristics. However, the budget could also be extended to wall-clock time or total monetary cost. The optimizer selects a subset $\mathcal{P}_B \subseteq \mathcal{P}$ with $|\mathcal{P}_B| \leq B$ and returns the approximate Pareto frontier $\hat{\mathcal{F}} = \text{Pareto}(\mathcal{P}_B)$. The optimization objective is then:

$$\max_{\mathcal{P}_B \subseteq \mathcal{P}} Q(\hat{\mathcal{F}}, \mathcal{F}) \quad \text{s.t.} \quad |\mathcal{P}_B| \leq B,$$

where $\mathcal{F} = \text{Pareto}(\mathcal{P})$ denotes the true (but unobservable) frontier. Here $Q(\hat{\mathcal{F}}, \mathcal{F})$ denotes the quality of the approximation. In practice, however, the true frontier $\mathcal{F}$ is unknown, so this formulation serves as a conceptual benchmark. Our evaluation in Sec. 5 instead assesses our optimizer empirically, by comparing the approximate frontiers $\hat{\mathcal{F}}$ it discovers against others.

In the following sections, we detail the two main components of the MOAR optimizer (depicted in Fig. 2): an expanded library of rewrite directives (Sec. 3) and a search algorithm that efficiently discovers high-quality pipelines within the evaluation budget (Sec. 4).

3 NEW OPERATORS AND DIRECTIVES

The design space of rewrites for semantic operator pipelines is theoretically infinite, spanning rewrites that target cost reduction (e.g., replacing LLM calls with code, fusing operators), accuracy improvement (e.g., decomposing complex tasks, chunking long documents), or both simultaneously. As in traditional query optimizers [16], there is no guarantee that the transformation rules are complete or sufficient for optimality; what matters is that the framework is extensible. MOAR’s optimization framework (Sec. 4) is designed independently of the specific directive library, so new directives can be incorporated as workloads evolve and models improve.

DocETL-V1 proposed three categories of rewrite directives targeting accuracy: *projection synthesis* (i.e., creating map operators that decompose tasks), *data decomposition* (i.e., splitting documents into chunks or groups of many documents into smaller-size groups), and *LLM-centric improvements* (e.g., prompting strategies) [44]. However, these directives typically increase cost. MOAR extends DocETL’s directive library in two ways. First, we introduce two new categories: *fusion and reordering* and *code synthesis*. Fusion combines multiple sequential operators into fewer ones, reducing the number of LLM calls—for example, two adjacent maps can be merged into a single map whose prompt instructs the LLM to perform both tasks and whose output schema is the union of both original schemas, or a map followed by a reduce can be absorbed into a single reduce when the map’s output does not generate the grouping keys. *Reordering* re-arranges commuting operators so that more selective operators run earlier. *Code synthesis* replaces LLM-powered operators with synthesized Python code, or inserts code-powered operations that preprocess data (e.g., regex-based

Table 1: Summary of new rewrite directives in MOAR (the LLM-Centric category is omitted here but appears in our technical report [56]). Categories marked with † are novel to MOAR. Directives marked with ‡ generate multiple candidate pipelines. The “visual” column qualitatively depicts each rewrite: block height reflects the length of the document in an LLM call, block width reflects complexity of the task performed by the LLM, and block depth represents the number of documents processed (shown only when it changes). Darker purple blocks indicate more expensive operators; lighter purple blocks indicate cheaper ones. Blocks with green borders are new or modified operators, and red borders denote unchanged operators that do less “work.”

Category	Directive	Transformation Pattern	Description	Visual
Fusion and Reordering†	❶ Same-type Fusion	$\text{map}_x \rightarrow \text{map}_y \Rightarrow \text{map}_z$; similarly for filter and reduce	Fuses pairs of same-type operators (map-map, filter-filter, reduce-reduce) into a single operator.	
	❷ Map-Reduce	$\text{map}_x \rightarrow \text{reduce}_{K,y} \Rightarrow \text{reduce}_{K,z}$	Combines the map and reduce into a single reduce. Applicable only when the output schema in x does not include the key(s) in K .	
	❸ Map-Filter	$\text{map}_x \rightarrow \text{filter}_y \Rightarrow \text{map}_z \rightarrow \text{code_filter}$	Expands the map to also compute the predicate produced by the downstream operator filter_y (so z 's output schema is the union of those from x and y), followed by a <code>code_filter</code> that simply checks the boolean attribute generated (in y 's schema).	
	❹ Filter-Map	$\text{filter}_x \rightarrow \text{map}_y \Rightarrow \text{map}_z \rightarrow \text{code_filter}$	Fuses filter and map logic into a single map. As in Map-Filter, the fused operation is “harder” or requires the LLM to do more “work;” thus it is wider in the visual.	
	❺ Reordering	$o_x \rightarrow o_y \Rightarrow o_y \rightarrow o_x$	Reorders commuting operators so that cheaper operators run earlier, akin to traditional operator reordering.	
Code Synthesis†	❻ Code Substitution	$o_x \Rightarrow \text{code_op}_x$	Replaces an LLM-powered operator with synthesized Python code.	
	❼ Code Sub. (Reduce)	$\text{reduce}_x \Rightarrow \text{code_reduce}_x \rightarrow \text{map}$	Splits a reduce into code-based aggregation plus a map that handles logic requiring an LLM and transforms the output into the schema specified in x . For example, a reduce that asks “generate a report of the most common themes in the documents” can be rewritten so <code>code_reduce</code> counts themes and concatenates relevant context, and the map generates the natural-language report.	
	❽ Doc. Compression (Code)‡	$o_x \Rightarrow \text{code_map} \rightarrow o_{x'}$	Uses synthesized Python code (e.g., with regexes) to deterministically extract only the relevant portions of the document, producing a shorter input for the downstream operator.	
	❾ Head/Tail Compression‡	$o_x \Rightarrow \text{code_map} \rightarrow o_{x'}$	Retains only the first h and last ℓ words (or lines) of each document via a synthesized <code>code_map</code> . Useful when key information typically appears at document boundaries (e.g., abstract, conclusion).	
Data Decomposition	❿ Chunk Sampling‡	$\text{split} \rightarrow \text{gather} \rightarrow \text{map} \rightarrow \text{reduce} \Rightarrow \text{split} \rightarrow \text{gather} \rightarrow \text{sample} \rightarrow \text{map} \rightarrow \text{reduce}$	Samples relevant chunks using BM25, embeddings, or random sampling. Reduces cost by processing only relevant chunks when full documents contain mostly irrelevant content.	
	⓫ Doc. Sampling‡	$\text{reduce}_{K,x} \Rightarrow \text{sample}_K \rightarrow \text{reduce}_{K,x}$	Samples a subset of documents within each group (e.g., using BM25, embeddings, or random sampling) before the reduce, reducing cost when groups contain many redundant or low-signal documents.	
	⓬ Cascade Filtering‡	$\text{filter}_x \Rightarrow \text{code_filter}^* \rightarrow \text{filter}_y \Rightarrow \text{filter}_x$	Inserts one or two cheaper “pre-filters” before filter_x . <code>code_filter</code> and filter_y (marked with * to indicate optionality; at least one is required) form a cascade in which each pre-filter removes additional documents before they reach filter_x .	
Projection Synthesis	⓭ Doc. Summarization	$o_x \Rightarrow \text{map} \rightarrow o_{x'}$	Produces a shorter version of each document by generating an LLM-written summary (via map) and passing that condensed text to the downstream operator.	
	⓮ Doc. Compression (LLM)‡	$o_x \Rightarrow \text{extract} \rightarrow o_{x'}$	Produces a shorter version of each document by generating an extract operator to return text spans from the original document; unlike the previous summarization rewrite, the output is a subset of the original text, not a transformation.	

extraction) before downstream LLM calls. Second, we adapt existing rewrite directive categories to also reduce cost—e.g., projection synthesis can compress documents to reduce tokens processed by downstream operators, data decomposition can limit processing to relevant chunks, and LLM-centric rewrites can substitute cheaper models. With DocETL-V1’s 13 directives, MOAR’s 18 new directives bring the total to over 30. All directives in Table 1 are new to MOAR; detailed descriptions and the typing and composability constraints they must satisfy are in App. B in our technical report [56]. We highlight a few examples below (directive numbers reference the corresponding row in Table 1):

- In **Fusion and Reordering**. The ❸ Map-Filter directive fuses a map followed by a filter into a single map whose prompt incorporates both the transformation and the filter predicate, producing a Boolean attribute that a downstream `code_filter` checks—reducing

cost by eliminating an LLM call per document. (❹ Filter-Map fusion into a Map is analogous but may not reduce cost when the filter has high selectivity.) These directives compose powerfully with others. For instance, in Ex. 1.1, if public defenders are only interested in defendants where firearm-related enhancement factors appear in both the police report *and* the charging summary, and the defender adds a filter to the pipeline in Ex. 1.1. MOAR can apply task decomposition (e.g., pipeline #10 in Fig. 1) to extract each factor type in parallel, reorder operations as per ❺ so the filter immediately follows the firearm-extraction map, and fuse them—avoiding downstream processing for irrelevant factors.

- In **Code Synthesis**. The ❽ Code-based Document Compression directive inserts a new `code_map` operator (in Python) to extract relevant portions of each document *before* the downstream LLM operator. Unlike ❻ Code Substitution (which replaces the LLM entirely), this directive preprocesses data to reduce document size. Our

key insight is that relevant content can often be identified via complex regular expressions or keyword matching—that LLM agents readily synthesize (e.g., 50–100 keyword variations for firearm-related content in Ex. 1.1). The downstream operator then runs on shorter documents, thereby lowering cost.

- In **Data Decomposition**. DocETL-V1 introduced chunking to handle documents exceeding LLM context limits. MOAR adds the **10** **Chunk Sampling** directive: after splitting, a sample operator selects relevant chunks (via BM25, embeddings, or random sampling) before the map. Analogously, the **11** **Document Sampling** directive selects a subset of documents before a reduce. These directives work well for tasks where processing all data is unnecessary—e.g., identifying common themes across thousands of customer reviews does not require reading every review, just a representative sample.
- In **Projection Synthesis**. MOAR extends projection synthesis with directives that *compress* documents rather than decompose tasks. **13** **Document Summarization** inserts a map generating an LLM-written summary of each document before downstream operators, reducing downstream operator costs.

To support the new directives, we introduce three new operator types. The **sample** operator selects a subset of documents (or chunks) most relevant to the downstream operator. It may use BM25 keyword search [39], embedding-based similarity, random sampling, or stratified variants of these methods that ensure each subgroup (e.g., based on metadata keys in the document) is proportionally represented in the sample. Next, the **extract** operator presents a document’s JSON representation with line numbers to an LLM, which returns ranges relevant to the operator’s natural language specification (e.g., “lines 45–67”); only those lines are retained, preserving the document’s key–value structure. Unlike a map that outputs text verbatim, **extract** guarantees exact subsets of documents and requires far fewer output tokens (thus reducing cost). Finally, code-powered operators (**code_map**, **code_reduce**, **code_filter**) execute synthesized Python instead of invoking an LLM. Overall, the aforementioned operators enable directives to reduce costs by processing smaller document portions (**sample**, **extract**) or replacing LLM calls with code. A complete operator list appears in Table 7 in our technical report [56]. Directives must also satisfy typing and composability constraints (formalized in App. B of our technical report [56]) that guarantee rewritten pipelines match the user’s expected output schema. However, there is no guarantee that rewritten prompts match the intent of the original, so the search algorithm evaluates each rewritten pipeline on a held-out sample. We discuss the soundness and completeness of our directive library, contrasting with the relational setting, in our technical report [56].

4 SEARCH ALGORITHM

In this section, we describe how MOAR efficiently searches over rewrite directives to discover high-quality and low-cost pipelines. Fig. 2 shows an overview of MOAR’s search algorithm. MOAR maintains state (shown in the dashed box) including model statistics, directive statistics, the current Pareto frontier, and a search tree rooted at the user-authored pipeline P_0 .

Search Space Representation. Unlike traditional query optimizers that construct plans from optimal subplans [16, 43], MOAR performs global search over complete pipelines. This design reflects

Table 2: Table of Notation

Category	Symbol	Description
Datasets & Pipelines	$D; D_O \subset D$	Dataset; sample to evaluate candidate pipeline on
	$P; P_0$	A candidate pipeline; the user-authored pipeline
	o^x	Operator with config $x = (p, s, m)$
	$p; s; m \in M$	Prompt template; schema; model
	$P \xrightarrow{r} P'$	Rewrite r transforms P into P'
Cost & Accuracy	d	A rewrite directive
	$c(P); a(P); \hat{c}(P); \hat{a}(P)$	Cost and accuracy of P ; empirical estimates on D_O
Search Tree	B	Evaluation budget
	$T_t = (V_t, E_t)$	Search tree with vertices V_t , edges E_t
	$V_t; \hat{\mathcal{F}}_t; \mathcal{F}$	Evaluated pipelines; frontiers
	$\text{children}(P); \text{parent}(P)$	Child/parent nodes
	$\text{desc}_t(P); \text{depth}(P)$	Descendants; depth
Pipeline Selection & Utility	$\text{last_action}(P)$	Directive used to generate P from $\text{parent}(P)$
	$A_t(P, c)$	Max accuracy at cost $\leq c$ excluding P
	$\delta_t(P)$	Vertical gain: $\hat{a}(P) - A_t(P, \hat{c}(P))$
	$n_t(P)$	Visit count
	$\bar{\delta}_t(P); U_t(P)$	Avg improvement; utility
Rewriting	$W(n_t(P))$	Max children P may generate
	$v(P, d)$	Usage count of d from P
	k	Number of candidate pipelines generated for a rewrite
	$\text{rank}_t(P)$	Accuracy rank of P compared to all pipelines explored

our observation from Sec. 1 that the best plan for a semantic operator pipeline may rely on subplans that are individually suboptimal, because the benefit of a subplan isn’t independent of its outputs. To support this global search, we represent the search space as a tree $T = (V, E)$ with the user-authored pipeline P_0 at the root. Each node $P \in V$ represents a *complete* pipeline configuration. Each edge $P \xrightarrow{r} P'$ applies a single rewrite r to produce child pipeline P' . Every node has exactly one parent and may have multiple children (shown as the search tree in Fig. 2). The path from P_0 to any node captures the sequence of rewrites used to construct it.

At each iteration t of the search algorithm, MOAR selects a node, applies a rewrite to generate a child pipeline, and *evaluates* it—that is, executes it on a small sample $D_O \subset D$ (e.g., 40 documents) to measure empirical cost $\hat{c}(P)$ and accuracy $\hat{a}(P)$. We write V_t for the set of pipelines evaluated after t iterations, E_t for the corresponding edge set, and $\hat{\mathcal{F}}_t$ for the Pareto frontier of V_t . Throughout this section, $\hat{c}(P)$ and $\hat{a}(P)$ denote sample estimates on D_O , not the population-level quantities $c(P)$ and $a(P)$ defined in Sec. 2.

Algorithm 1: MOAR Search

Input: User pipeline P_0 , dataset D , sample $D_O \subset D$, model pool M , budget B
Output: Pareto frontier $\hat{\mathcal{F}}_B$

```

1 Function MOAR( $P_0, D_O, M, B$ ):
  // Initialization: evaluate  $P_0$  with all models
2  $T \leftarrow \text{Initialize}(P_0, D_O, M)$ ;
3 initialize directive usage map  $v(P, d) \leftarrow 0$  for all  $P, d$ ;
4  $t \leftarrow |V|$ ; // Number of pipelines evaluated so far
  // Main search loop
5 while  $t < B$  do
  // Selection: traverse tree to find node to rewrite
6  $P^* \leftarrow \text{Select}(P_0, T_t)$ ;
  // Rewriting and evaluation (parallelized)
7  $(P', r, \hat{c}(P'), \hat{a}(P'), k) \leftarrow \text{RewriteAndEvaluate}(P^*, G_t, v, D_O)$ ;
  // Update tree with new pipeline and rewrite edge
8 add  $P'$  to  $V$  and edge  $P^* \xrightarrow{r} P'$  to  $E$ ;
9  $t \leftarrow t + k$ ; // Increment by number of candidates evaluated
10 recompute  $\hat{\mathcal{F}}_t$  from  $V$ ;
11 end
12 return  $\hat{\mathcal{F}}_B$ ;

```

Algorithm Overview. MOAR searches for better pipelines by iteratively selecting, rewriting, and evaluating candidates, as described in Algorithm 1 and illustrated in Fig. 2. The search process begins with *initialization*, where MOAR constructs an initial frontier

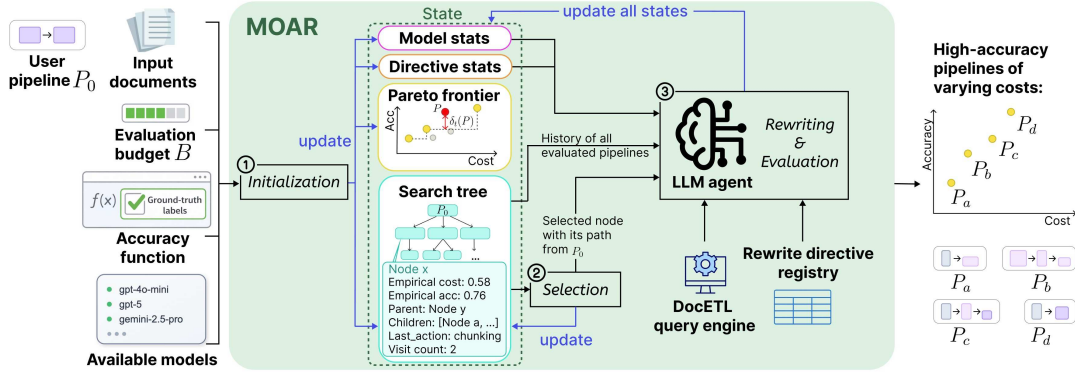


Figure 2: System architecture of MOAR. The optimizer takes as input a user pipeline P_0 , input documents, user-defined evaluation budget, user-authored accuracy function, and available models. In the initialization phase (①), MOAR evaluates P_0 with all models to construct an initial frontier. The main search loop alternates between selection (②)—choosing which pipeline to rewrite based on contribution of it and its descendants to the Pareto frontier—and rewriting & evaluation (③)—using an LLM agent to select and instantiate a rewrite directive from the registry, then executing the resulting pipeline. The output is a set of high-accuracy pipelines spanning different costs.

that provides diverse starting points across the accuracy–cost spectrum. After initialization, each iteration of the search loop proceeds through two phases. First, in the *selection phase*, MOAR adapts a multi-armed bandit framework [6] to decide which pipeline to rewrite next. Unlike traditional UCT, where each node’s reward is independent, a node’s reward in our setting depends on all other nodes because the Pareto frontier is a global function of every pipeline explored; we therefore define a custom utility metric based on marginal accuracy contribution that handles this interdependence. There are also infinitely many possible rewrites, so we limit the branching factor and delegate action selection to an LLM agent that reasons about pipeline semantics to prioritize which directives to try. Second, in the *rewriting and evaluation phase*, the LLM agent chooses and instantiates a rewrite directive to produce a child pipeline. The child P' is then executed on D_o to obtain $(\hat{c}(P'), \hat{a}(P'))$, which informs selection in future iterations. Each iteration incurs significant latency due to LLM-guided rewriting and pipeline execution. MOAR therefore parallelizes search across multiple workers, with only the selection phase synchronized to ensure consistency. The loop repeats until the evaluation budget B is exhausted, returning the final frontier $\hat{\mathcal{F}}_B$.

Next, we describe each phase in detail: initialization (Sec. 4.1), selection (Sec. 4.2), and rewriting and evaluation (Sec. 4.3). Table 2 summarizes notation used.

4.1 Initialization

Before the main search loop begins, MOAR initializes two components: the Pareto frontier and the state used throughout search.

Frontier Initialization. Building an initial frontier prevents the search from getting trapped early in local minima and provides statistics that aid future selection decisions. Given the user-authored pipeline P_0 configured with a default model m_0 (e.g., gpt-4o-mini), MOAR evaluates P_0 with all models in the pool M (e.g., gpt-4o-mini, gpt-4o, gemini-2.5-flash).³ In other words, for each model $m_i \in M$, MOAR creates a pipeline variant by substituting m_i into all operators of P_0 , then measures its cost and accuracy $(\hat{c}, \hat{a})$ on D_o . These model variants become children of P_0 in the search tree, yielding frontier $\hat{\mathcal{F}}_{|M|}$. Next, for each pipeline in $\hat{\mathcal{F}}_{|M|}$, MOAR

uses an LLM agent to generate exactly two rewrites—one targeting accuracy improvement and one targeting cost reduction—using our standard rewriting procedure (to be described in Sec. 4.3). By spawning grandchildren only from frontier children ($\hat{\mathcal{F}}_{|M|}$) rather than all pipelines in $V_{|M|}$, we limit the budget consumption from initialization. At the end of initialization, we disable non-frontier model variants from future selection, ensuring subsequent iterations focus on promising regions of the search space. At the end of initialization (step ① in Fig. 2), the search tree contains P_0 at the root, model variants as children, and rewrites of frontier pipelines as grandchildren.

Other State Initialization. For each node P , MOAR maintains:

- (i) $\hat{c}(P)$ and $\hat{a}(P)$: empirical cost and accuracy on D_o ;
- (ii) $\text{parent}(P)$ and $\text{children}(P)$: encoding tree structure;
- (iii) $n_t(P)$: visit count, equal to $1 + |\text{desc}_t(P)|$ where $\text{desc}_t(P)$ denotes the descendants of P in V_t ; and
- (iv) $\text{last_action}(P)$: the directive applied to generate P from $\text{parent}(P)$.

Most statistics remain fixed throughout search; only $\text{children}(P)$ and $n_t(P)$ evolve as new pipelines are evaluated. Additionally, MOAR initializes aggregate statistics. (i) *Model statistics* record the cost and accuracy achieved by each model variant of the original pipeline (i.e., each child of P_0), providing a controlled comparison across models on identical operators; this helps agents select models when synthesizing new operators. (ii) *Directive statistics* record, for each directive d , the average change in cost and accuracy induced by applying d , measured as the difference between a pipeline’s metrics and those of its parent. Agents receive both model and directive statistics as context during rewriting (Sec. 4.3).

4.2 Selection

After initialization, the search tree contains $|V_t| = |M| + 2|\hat{\mathcal{F}}_{|M|}|$ pipelines (counting toward the budget B). Now, in each iteration, the selection phase identifies which pipeline P^* to rewrite—equivalently, which path in the search tree to extend by one more rewrite. The selector must balance exploitation of high-performing paths with exploration of under-tested ones. MOAR achieves this balance by assigning each pipeline a multi-armed bandit-style utility score [6].

Utility Function. We begin by formalizing utility. Let

$$A_t(P) := \max\{\hat{a}(P') : P' \in \text{Pareto}(V_t \setminus \{P\}), \hat{c}(P') \leq \hat{c}(P)\}$$

³If $|M| > C_m$ (set to 12 in our implementation), we subsample up to 3 models per family (e.g., gpt-4.1-nano, gpt-4.1-mini, gpt-4.1) from randomly selected families.

be the highest accuracy achievable at cost $\hat{c}(P)$ or lower, excluding P itself. The *contribution* of pipeline P to the frontier is then $\delta_t(P) := \hat{a}(P) - A_t(P)$, measuring how much P improves accuracy beyond other pipelines with comparable cost. When $\delta_t(P) > 0$, pipeline P extends the frontier; otherwise it is dominated. This quantity $\delta_t(P)$ is visualized as the vertical distance between the red point and the frontier in the “Pareto frontier” panel of Fig. 2.

One widely used strategy for multi-armed bandits is the Upper Confidence Bound (UCB) algorithm [6]. However, we cannot apply UCB directly because the pipelines are not independent; each is derived from rewrites of its ancestors. Instead, we use UCT [25], which extends UCB to a tree-structured search space. Following UCT, we define the utility score as:

$$U_t(P) = \underbrace{\frac{\delta_t(P) + \sum_{P' \in \text{desc}_t(P)} \delta_t(P')}{n_t(P)}}_{\text{exploitation}} + \underbrace{\sqrt{\frac{2 \ln n_t(\text{parent}(P))}{n_t(P)}}}_{\text{exploration}}, \quad (1)$$

where $\text{desc}_t(P)$ denotes all descendants of P in V_t .

The *exploitation* term in Eq. (1) measures the average frontier contribution along the subtree rooted at P : we sum δ_t over P and all its descendants, then divide by the visit count $n_t(P) = 1 + |\text{desc}_t(P)|$. Unlike standard UCB, which tracks rewards for independent arms, UCT aggregates rewards across an entire subtree because a node’s value depends on what rewrites become reachable after selecting it. Then, the *exploration* term encourages trying under-visited nodes. Standard UCB uses the total number of iterations across all nodes in the numerator; UCT instead uses the parent’s visit count $n_t(\text{parent}(P))$ because this represents how many times we have had the opportunity to even select P for rewriting. A child with few visits relative to $n_t(\text{parent}(P))$ is under-explored compared to its siblings and receives a higher exploration bonus.

Hierarchical Traversal with Progressive Widening. Because utility scores depend on $n_t(\text{parent}(P))$, they are only comparable among siblings. We therefore traverse the tree top-down to select P^* : starting at P_0 , we repeatedly select the child with the highest utility until reaching a node P^* that can generate a new child. In standard UCT [25], when a node is selected, *all* possible rewrites (i.e., actions) are applied before the next iteration of selection. This is feasible for small action spaces (e.g., board games), but problematic here since a single pipeline could spawn hundreds of children ($30+ \text{directives} \times \text{multiple operators} \times \text{multiple instantiations}$). To limit branching, we use *progressive widening*, a technique from Monte Carlo Tree Search for large action spaces [10]. The idea is to cap the number of children based on visit count, allowing the action space to be explored gradually. We accordingly set the maximum number of children for node P to $W(n_t(P)) = \max(2, 1 + \sqrt{n_t(P)})$, where the $\sqrt{n_t(P)}$ growth rate is typical [10]. For example, a node with four visits may have at most three children; only after nine visits may it produce a fourth. This sublinear growth forces exploration of deeper paths before generating more rewrites from any single node. Algorithm 2 in App. C in our technical report [56] specifies the complete procedure.

4.3 Rewriting and Evaluation

Given the selected pipeline P^* , the rewriting phase generates a new child pipeline P' by applying a directive. In DocETL-V1 [44], optimization enumerates all applicable directives for each operator

(or operator prefix) in the pipeline, then invokes an LLM agent to instantiate each one. The exhaustive approach is infeasible for MOAR, because the rewrite space is too large. Instead, MOAR delegates the entire rewriting decision to an LLM agent, which can reason about pipeline semantics to choose which directive to apply, which operators within the pipeline to target, and how to instantiate the directive. Algorithm 3 in App. C in our technical report [56] details the rewriting and evaluation procedure. Unlike selection, rewriting and evaluation are fully parallelized—multiple workers can simultaneously rewrite different selected pipelines and execute them on D_o . We now describe how directives are encoded, how the agent chooses and instantiates them, and how resulting pipeline(s) are evaluated.

4.3.1 How Directives Are Encoded Each rewrite directive is defined by a Python class that encapsulates documentation for the LLM agent and execution logic. The documentation includes:

- **Name and descriptions.** A unique identifier (e.g., `code_sub`), the LHS/RHS pattern (e.g., `reduce  $\Rightarrow$  code_reduce  $\rightarrow$  map`), and a plain-language explanation.
- **Use case guidance.** A natural language explanation of when to apply the directive and what scenarios benefit most from it. These are manually curated based on experience from our deployment of DocWrangler, an IDE for authoring DocETL pipelines [45], and do not overlap with pipelines in our evaluation (Sec. 5). The full directive documentation is available in the [codebase](#).
- **Instantiation schema.** A Pydantic model [12] specifying parameters needed to apply the directive (e.g., `clarify_instructions` requires a `clarified_prompt`), with optional validators to enforce constraints (e.g., that the new prompt preserves all input variables).
- **Example application.** A concrete example showing the original pipeline configuration, the instantiation parameters that would be generated, and the resulting transformed pipeline.

Each directive also implements dynamic execution logic through two methods: `instantiate()` and `apply()`. The `instantiate()` method generates the parameters needed to apply the directive to a target pipeline, described in detail below. The `apply()` method takes the instantiated parameters (the structured object produced by `instantiate()`) and the current pipeline, and produces the rewritten pipeline to be parsed and executed by DocETL.

4.3.2 Choosing and Instantiating Directives Before the agent chooses a directive, we prune the set of directives to filter out redundant or trivial rewrites. First, we prune *cycles*—rewrites that reverse a transformation applied earlier on the path from P_0 to P^* . Specifically, we prune: (i) a chaining directive immediately followed by a fusion directive, and (ii) applying a model substitution directive at a first-layer node, which only switches back to previously tried models. Second, we prune *no-ops*—rewrites that redundantly apply the same type of transformation. We prune: (i) applying a chunking directive to a pipeline that already uses chunking, and (ii) consecutive compression or summarization directives that attempt to reduce already condensed content.

Our agent-based rewriting proceeds in two steps. First, the agent chooses which directive to apply and which operators to target, seeing only directive names, descriptions, and use case guidance. Second, the agent loads the full instantiation schema and example application to generate concrete parameters. This *progressive*

disclosure—a technique in user interface design for reducing cognitive load by revealing information gradually [9]—avoids overwhelming the agent with details.

Choosing a Directive. The agent receives as input (via its prompt):

- Pipeline P^* (as YAML).
- For each directive after pruning: its name, description, and use case guidance.
- A list of all rewrite paths explored so far, along with corresponding $\hat{c}$ and $\hat{a}$. One rewrite path might look like: `ROOT → model_sub(gpt-4.1) → doc_chunking(size=1000)` (cost: \$4.07, acc: 0.739).
- The path of rewrites from P_0 to P^* and its depth (P^*).
- Model and directive statistics, as defined in Sec. 4.1.
- An objective—either “reduce cost while preserving accuracy” or “improve accuracy”—determined by P^* ’s rank among evaluated pipelines. If $\text{rank}_t(P^*) \leq |V_t|/2$ (where rank 1 is most accurate), the objective is cost reduction; otherwise, accuracy improvement. Providing different objectives to the agent helps discover a diverse Pareto frontier spanning different accuracy-cost trade-offs.

The agent chooses one directive d whose LHS matches a subsequence of operators in P^* and identifies which operators to rewrite.

Instantiating the Directive. Once chosen, the agent loads the directive’s full documentation (e.g., instantiation schema, example application) and generates concrete parameters to produce a specific rewrite $P^* \xrightarrow{r} P'$. MOAR invokes the directive’s `instantiate()` method. Instantiation is an interactive loop. The agent receives: (i) a system prompt establishing its role (e.g., “expert at optimizing LLM-powered data processing pipelines”), (ii) the directive’s full documentation (instantiation schema and example application) along with the target pipeline P^* and target operators, and (iii) access to sample documents from D_o . At each step, the agent can call `read_next_doc()` to inspect samples—e.g., to identify appropriate chunk sizes or detect patterns informing prompt refinements—or output a structured object matching the instantiation schema. MOAR validates outputs against the schema; if validation fails, the error is returned for refinement. The loop continues until valid or a retry limit is reached. Validated parameters are passed to `apply()`, producing P' . For directives with parameters that are difficult for LLMs to select—e.g., chunk sizes in document splitting—MOAR generates multiple instantiations with different parameter values, evaluates each on D_o , and picks the highest-accuracy instantiation. Six of the 18 new directives are parameter-sensitive (marked with ‡ in Table 1); App. B in our technical report [56] details the number of candidates for each and references the agent prompts used for directive selection and instantiation.

4.3.3 Evaluation and Error Handling Once a valid rewrite P' is generated, MOAR executes it on D_o to obtain $(\hat{c}(P'), \hat{a}(P'))$, reusing cached measurements for identical pipelines. MOAR handles errors—e.g., rewrites producing invalid YAML or malformed operator configurations, or LLM API failures—by retrying or discarding the pipeline and adjusting visit counts accordingly (details in our technical report [56]). Once the budget B is exhausted, MOAR outputs the final Pareto frontier $\hat{\mathcal{F}}_B$.

5 EVALUATION

We evaluate MOAR across a diverse set of workloads spanning legal, biomedical, government, consumer, and corporate domains. Compared to baselines, **MOAR discovers the highest-accuracy**

pipeline on every workload. On average, MOAR achieves 27% higher accuracy than the next-best optimizer, ABACUS [41], while matching its best accuracy at only 54.5% of its cost. We first describe our setup (Sec. 5.1). We then present the accuracy improvements and cost savings achieved by MOAR (Sec. 5.2). Finally, we examine the characteristics of high-accuracy pipelines discovered by MOAR (Sec. 5.3). Our experiment artifacts are released at [this Google Drive link](#). The full agent prompts used for directive selection and instantiation are available in our codebase: the [main agent logic](#) and [individual directive prompts](#).

5.1 Setup

We run MOAR with a budget of 40 pipeline evaluations per workload, using gpt-5 as the agent for instantiating rewrite directives. MOAR selects models from a pool of 11 LLMs, including gpt-4o-mini, gpt-4o, 3 gpt-4.1 variants, 3 gpt-5 variants, and 3 gemini-2.5 variants. All models are available to all optimizers. Unless otherwise noted, the user-specified or initial pipelines prior to optimization use gpt-4o-mini, as in prior work [36, 41, 44].

DocETL comprises over 30,000 lines of Python code, of which MOAR accounts for approximately 16,000 lines. The directive library alone requires over 9,000 lines (each directive requires 300–600 lines). We use LiteLLM as a wrapper around Google Gemini and Azure OpenAI APIs. All experiments were executed on Modal, a cloud computing platform.

5.1.1 Baselines We compare MOAR against 4 baseline systems and 1 ablation. We compare against open-source systems that support a semantic map operator. For each baseline system, we express the pipeline using the baseline’s interface, minimally modifying the DocETL operators so the pipeline can be parsed by the baseline system. All baselines except DocETL-V1 support selecting from multiple models; DocETL-V1 uses only gpt-4o-mini.

- **DocETL-V1** [44]. We run the original DocETL optimizer, which optimizes only for accuracy and returns a single plan. Unlike all other baselines, DocETL-V1 does not access ground-truth labels; it relies solely on an LLM judge to select plans.

- **LOTUS** [36]. We express each pipeline using LOTUS’s semantic operators. LOTUS does not support structured output schemas, so we minimally edit prompts to request JSON-formatted responses and write custom Python code to parse LOTUS outputs. LOTUS performs cost reduction for filters, joins, and group-bys by swapping in cheaper models (i.e., gpt-5-nano). Like DocETL-V1, LOTUS always returns a single optimized plan.

- **Palimpsest/ABACUS** [30, 41]. We express each pipeline using Palimpsest (PZ)’s [30] operators and use its ABACUS optimizer [41]. ABACUS does not directly return a Pareto frontier; instead, it allows users to specify a cost budget and returns the maximum-accuracy plan within that budget. Following discussion with the authors, we construct a Pareto frontier by running PZ with: (i) no budget constraint (to obtain the highest-accuracy plan), and (ii) budgets set to 50%, 25%, and 10% of the unconstrained plan’s cost. As of writing this paper (October 15, 2025), PZ does not support LLM-powered reduce operators, so we omit it from workloads requiring the DocETL reduce operator.

- **Simple agent (SA).** We test whether an LLM agent, that can use DocETL as a tool and read DocETL-V1’s rewrite directives in the codebase, can discover effective pipelines. We provide a gpt-5

Table 3: Summary of evaluation workloads.

Workload	#Docs (avg wds)	Task (init. pipeline)	Metric
CUAD	510 (7.7k)	Extract clause spans from contracts (map)	Clause-type F1
Game Reviews	(97.7k)	Rank positive & negative reviews chronologically (map)	Hallucination, sentiment, ordering
BlackVault	733 (7.4k)	Classify event types; recall locations per type (map→reduce)	Location recall
Biodex	(71.2k)	Link papers to adverse drug reactions (map)	Rank-precision@5
Medec	3,848 (147)	Detect & correct errors in clinical notes (map)	Detection F1, correction similarity
Sustainability	5,436 (38.7k)	Filter reports; classify sector; summarize initiatives (filter→map→reduce)	Sector accuracy, name accuracy

Table 4: Best accuracies. Highest per workload is bolded; next-best is underlined. Last row shows MOAR’s average relative gain. For Biodex, LOTUS and PZ include two variants (d, r&r). For BlackVault, the first row reports raw recall; the second row is normalized.

Workload	DocETL-V1	LOTUS	PZ	SA	MOAR (no search)	MOAR
CUAD	0.471	0.402	<u>0.694</u>	0.521	0.495	0.762
Game Reviews	0.608	0.522	<u>0.683</u>	0.467	0.703	0.804
BlackVault	5.339	3.020	–	7.230	4.055	37.333
(normalized)	(0.143)	(0.081)	–	(0.194)	(0.109)	(1.000)
Biodex	0.247	0.260 (d)	0.260 (d)	<u>0.333</u>	0.259	0.370
Medec	0.534	0.538	0.536	0.726	0.755	<u>0.742</u>
Sustainability	0.632	0.516	–	0.543	0.560	0.646
Avg Gain (%)	+130.71%	+209.53%	+26.65%	+94.36%	+157.04%	–

agent with three tools: (i) reading sample documents, (ii) reading the original paper [44], codebase, and [DocETL documentation](#), and (iii) executing any pipeline on samples and observing its accuracy and cost. The agent proposes rewrites until its context window is exhausted ($\approx 400k$ tokens) or it calls a “done” tool to indicate completion. Then, of all pipelines generated by the agent, we retain the Pareto frontier.

• **MOAR (no search).** Here, we run SA but also provide descriptions of all 18 new MOAR rewrite directives—to serve as a true ablation of MOAR’s search algorithm. As with SA, we retain the Pareto frontier over all pipelines produced.

5.1.2 Workloads We evaluate across six workloads spanning legal, gaming, intelligence, biomedical, medical, and sustainability domains (Table 3). Four are from prior work [36, 41, 44]; two (Medec, Sustainability) are new, inspired by real DocETL users. Metrics with multiple components are averaged when we report results in Table 4. Full descriptions of the workloads are in our technical report [56]. The initial pipeline for each workload uses a minimal number of semantic operators, representing what a user would naturally write when first encountering the problem [45].

Each workload uses a dataset D , split into D_o (40 documents, for optimization) and a held-out test set $D_T = D \setminus D_o$ (100 documents). Optimizers search over candidate pipelines using D_o , and all reported results reflect execution on the held-out test set D_T .

5.2 Results

We present main results on accuracy (Sec. 5.2.1), cost savings and Pareto frontier quality (Sec. 5.2.2), and optimization overhead (Sec. 5.2.3), and additional experiments and ablations in Sec. 5.2.4.

5.2.1 Accuracy Improvement Table 4 summarizes MOAR’s accuracy improvements across all baselines. **MOAR achieves the highest accuracy on every workload among all baselines, with gains over the next most accurate baseline ranging from +2.2%**

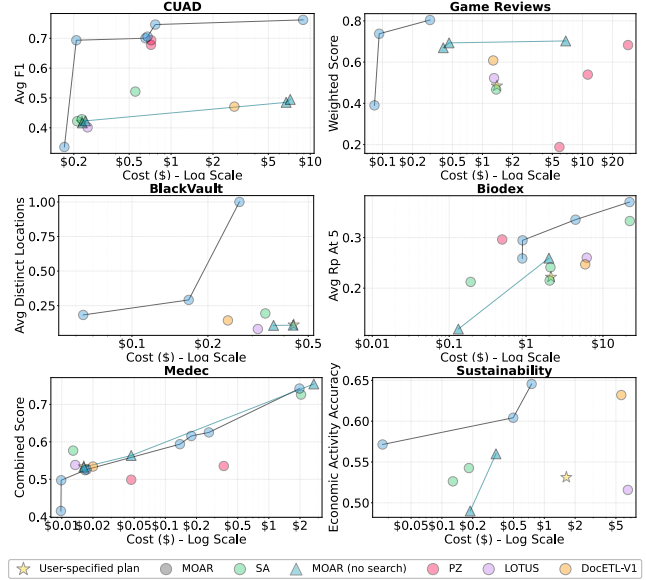


Figure 3: Pareto frontiers for each method, for each held-out test set. “User-specified plan” is the initial pipeline authored by the user.

(Medec, Sustainability) to +415.5% (BlackVault). MOAR’s pipelines are also structurally more complex than those produced by the baselines. **MOAR’s highest-accuracy pipelines use, on average, 2.3× as many operators as the baseline pipelines.** MOAR achieves the highest accuracy on every workload except Medec, where MOAR (no search) edges it out (0.755 vs. 0.742) by applying five rewrites: four of our predefined directives (model substitution, code-based document compression, gleaning, and prompt rewriting) and one the agent wrote from scratch (a code map to programmatically clean the output). SA also performs well by rewriting the prompt and switching to gpt-5. Just switching to gpt-5 achieves 8% lower accuracy at 1.25× the cost and does not appear on the Pareto frontier. On Biodex, RP@5 is low across all methods; an error analysis in our technical report [56] finds that most of the gap is attributable to errors in the ground truth rather than model performance.

5.2.2 Cost Savings and Pareto Frontier Quality Table 5 shows MOAR’s cost to match each baseline’s highest accuracy. App. D in our technical report [56] provides pairwise cost comparisons between all methods. On average, MOAR finds plans that **match the simple agent’s best accuracy with 0.436× the cost, LOTUS’s with 0.487×, PZ’s with 0.545×, and DocETL-V1’s with 0.256× the cost.** The largest cost savings occur on Game Reviews, where MOAR achieves PZ’s highest accuracy with 0.003× the cost.

There are two cases where MOAR does not achieve cost savings for top-accuracy baseline plans. First, for Medec, MOAR costs 1.245× LOTUS to match its accuracy. However, LOTUS’s accuracy on Medec is 53.8%—nearly 20 percentage points worse than MOAR’s maximum (74.2%)—so returning plans at such low accuracy may not be useful in practice. Second, for Biodex, MOAR costs 1.840× to match PZ-r&r’s accuracy. However, MOAR still finds a plan that is more accurate, albeit more expensive.

Biodex also illustrates how different logical plans yield unpredictable performance: LOTUS-d (direct) outperforms LOTUS-r&r (retrieve-and-rerank), while PZ-r&r outperforms PZ-d. MOAR discovers its highest-accuracy pipeline through chunking and sampling rewrites, which resembles PZ-r&r’s retrieval-based approach.

Interestingly, the simple agent achieves the second-best accuracy on Biodex by simply using gpt-5 on the single-map pipeline, outperforming all LOTUS and PZ variants.

Finally, Fig. 3 shows the Pareto frontier for each method. MOAR completely dominates all other methods, and the MOAR (no search) ablation, on CUAD, Game Reviews, BlackVault, and Sustainability. On Biodex and Medec, only two baseline pipelines in each of these two workloads are not dominated. For Medec, both non-dominated pipelines achieve lower accuracy than the original user-specified pipeline, limiting their practicality. For Biodex, only one baseline pipeline (PZ’s highest-accuracy plan) is not dominated by MOAR.

5.2.3 Pareto Optimization Overhead In Table 6, we report optimization costs and latencies, as well as expected execution costs at 1-million document scale. MOAR and ABACUS (PZ) are the only optimizers that construct a Pareto frontier. Relative to PZ, MOAR discovers substantially more accurate pipelines (27% higher accuracy on average) and more cost-efficient pipelines (achieving PZ’s accuracy at 0.545× the cost). This comes with a higher optimization cost: MOAR’s optimization cost is 2.11× that of ABACUS, though its optimization latency is only 0.562× that of ABACUS. Users are often willing to incur substantial optimization costs to achieve highest accuracies [23]. Moreover, since optimization is a one-time cost that amortizes over repeated pipeline executions, a high cost can be acceptable if it lowers execution costs at scale [2, 47].

5.2.4 Additional Experiments We conduct four additional experiments, with full details in App. D of our technical report [56]; we summarize the results here.

- To determine whether MOAR’s gains come primarily from selecting better models, we run the user-specified pipeline with every model in our pool and construct the resulting Pareto frontier for each workload. MOAR achieves higher accuracy on every workload: on Game Reviews, BlackVault, and Sustainability, MOAR’s best accuracy is 1.3–8.6× higher; on CUAD, Biodex, and Medec, the accuracy is within five percentage points. Regarding cost, MOAR matches the model-substitution frontier’s best accuracy at 0.009–0.435× the cost on four workloads, and still at lower cost on the remaining two (Biodex, Medec).
- To understand how much the new search algorithm and the expanded directive library each contribute to MOAR’s performance, we evaluate MOAR with only DocETL-V1’s directives (which we call MOAR-V1; see Fig. 6 in our technical report [56]). On four of six workloads, MOAR produces strictly better Pareto frontiers than MOAR-V1; on the remaining two, MOAR-V1 dominates DocETL-V1. Overall, the search algorithm and the expanded library are both necessary for good performance on all workloads.
- To evaluate whether open-source LLMs can replace gpt-5 as the optimization agent, we evaluate four open-source LLMs on CUAD (see Fig. 5 in our technical report [56]) and find that two (Kimi K2 Thinking [24], Llama-4-Maverick [34]) produce comparable Pareto frontiers while two smaller LLMs fail due to insufficient context or instruction-following capabilities.
- To understand whether accuracies on the optimization sample generalize, we report accuracy differences between D_o and D_T across all Pareto-optimal pipelines (see Table 10 in our technical report [56]); the average difference is less than 5 percentage points.

Table 5: Cost of the cheapest MOAR plan that matches or exceeds each baseline’s best accuracy, as a multiple of that baseline’s cost. “–” denotes that the baseline does not reach the original accuracy; “n/a” that it is not evaluated on the workload. Reported savings exclude one-time optimization costs.

Workload	DocETL-V1	LOTUS	PZ	SA	MOAR (no search)
CUAD	0.073×	—	0.290×	0.377×	0.029×
Game Reviews	0.072×	0.071×	0.003×	—	0.014×
BlackVault	0.267×	—	n/a	0.497×	—
Biodex	0.152×	0.145×	1.840×	0.196×	0.451×
Medec	0.840×	1.245×	0.046×	0.966×	0.999×
Sustainability	0.133×	—	n/a	0.143×	0.078×

Table 6: Optimization Overhead and Expected Execution Costs. MOAR’s cost multiple relative to the initial plan (× Init.) is shown in parentheses. Expected execution costs are computed by scaling the most accurate pipeline’s cost to the full dataset and to 1M documents. “–” indicates the method is not evaluated on that dataset.

	Opt. Cost (\$)				Opt. Latency (s)			Exec. Cost (\$)	
	MOAR	SA	PZ	DocETL V1	MOAR	SA	PZ	Full Data	1M
CUAD	44.0 (36.0×	0.34	54.9	1.58	5,536	157	11,579	45.3	88.9k
Game Reviews	57.3 (44.0×	0.29	28.8	6.60	8,382	2,502	19,718	28.0	2.99k
BlackVault	35.4 (11.0×	0.24	—	1.84	2,353	1,092	—	1.95	2.67k
Biodex	188 (0.45×	8.94	93.2 (d) 6.26 (r&r)	14.4	3,908	756	9,711 (d) 1,768 (r&r)	4.31k	222k
Medec	16.7 (21.7×	0.27	4.59	0.01	2,889	166	3,066	72.9	19.0k
Sustainability	79.2 (0.88×	2.59	—	43.7	2,070	547	—	40.9	7.52k

5.3 Insights from MOAR’s Pipelines

To understand what characterizes accurate pipelines, we analyze the 5 most accurate Pareto-optimal pipelines per workload (29 total, since Game Reviews has only 4 pipelines on the frontier).

- **86% use a modified logical plan.** Top pipelines exhibit different logical structures—adding, removing, or restructuring operators. For example, in BlackVault, the initial pipeline first extracted event types via a map, then aggregated locations per event type using a reduce that reprocesses all documents. MOAR rewrote the initial map to extract both event types and locations, so the downstream reduce simply combines and deduplicates pre-extracted lists rather than re-analyzing full documents.
- **48% use agent-authored code.** These pipelines incorporate agent-authored code operations to replace or complement LLM-powered components. In a top-accuracy CUAD pipeline, a code operation using regex pattern matching was inserted before the LLM-powered map to identify clause-relevant sections and extract fixed-size context windows around each match. This reduced cost by 48.76% while improving accuracy by 3.1%.
- **11 distinct rewrite directives are selected, but a few dominate.** Model substitution is the most frequent (25 usages) but *never* appears alone. Every pipeline combines it with at least one other rewrite directive. The most common non-model-substitution directives are clarify instructions (13 usages), chunking (11), code-based document compression (9), and head/tail compression (8), spanning LLM-centric improvements, data decomposition, and code synthesis categories. Fusion and reordering directives appear 4 times.
- **High-accuracy pipelines are discovered late.** Among the 29 pipelines, 51.72% were found after iteration 20 (the second half of the 40-iteration search), and 34.48% were discovered after iteration

	Rule-based	Agentic
Data-indep.	e.g., model substitution, ensembling, operator reordering ABACUS [41], FLOCK-MTL [13], UNIFY [55], DOCDB [28], MOAR	e.g., operator fusion, task decomposition DOCETL-V1 [44], CAESURA [51], UNIFY [55], AOP [54], MOAR
Data-dep.	e.g., model cascades, fine-tuning, RAG ABACUS [41], LOTUS [36], THALAMUSDB [21], CORTEX-AISQL [2], ELEET [52], UNIFY [55], DOCDB [28], MOAR	e.g., code synthesis, other rewrite directives DOCETL-V1 [44], CORTEX-AISQL [2], MOAR

Figure 4: Space of rewrites for semantic operator pipelines. Data-independent rewrites can be instantiated without sample data; data-dependent rewrites require samples to learn configurations or synthesize transformations. Systems shown in purple.

30—demonstrating that MOAR avoids premature convergence and maintains strong exploration throughout.

• **Most explored pipelines never contribute to the Pareto frontier.** Across workloads, 56–79% of explored pipelines are neither on the final Pareto frontier nor ancestors of a frontier point in the search tree (58% on CUAD, 79% on Game Reviews, 76% on BlackVault, 64% on Biodex, 56% on Medec, and 59% on Sustainability). We discuss strategies for reducing this wasted budget in Sec. 7 of our technical report [56].

The empirical patterns above suggest some guidelines for pipeline authors. First, *invest in specification quality*—the logical plan will likely change during optimization, but a well-specified prompt helps the optimizer produce rewrites that preserve the user’s intent. For example, on Medec, the carefully specified prompt from Abacha et al. [1] allowed MOAR to find a near-optimal plan early in search, whereas workloads with vaguer prompts required more iterations to converge. Second, *don’t default to the highest-accuracy plan*: the accuracy-cost trade-offs are steep (on CUAD, a 2.07% accuracy drop reduced cost by 91.34%). Users with budget constraints can often find plans that are nearly as accurate at a fraction of the cost.

6 RELATED WORK

We discuss semantic data processing systems and optimization.

Semantic Data Processing Systems. Systems that expose AI-powered data processing capabilities have proliferated over the past two years. ThalamusDB [21] was among the earliest to support natural language filters and joins in SQL. LOTUS [36] coined the term “semantic operators” for this type of LLM-powered data processing; other semantic data processing systems include Palimpsest [30], Aryn [4], DocETL [44], FlockMTL [13], ELEET [52], Unify [55], DocDB [28], and Stretto [42]. Other systems target specific applications, e.g., templated documents [29], data extraction [5, 49], and social science [20]. Industrial DBMSes from Databricks [57], DuckDB [14], Snowflake [2, 48], Google BigQuery [18], and Google AlloyDB [15] also now support LLM-powered UDFs.

Semantic Query Optimization. Query optimizers have three components: a plan space, a cost model, and a search algorithm.

(1) **Plan space.** Classical relational database optimizers search over rewrites such as filter and join reordering [11, 19, 43]. These are rule-based, algebraic transformations. Semantic operators admit a richer space of rewrites—including transformations over the meaning of the task and data—which we organize along two axes (Fig. 4): whether rewrites are *data-dependent* (requiring samples to instantiate) or *data-independent*, and whether rewrites are *agentic* (synthesized by an LLM, as in DocETL) or *rule-based* (not requiring an LLM agent to instantiate). These quadrants vary in instantiation cost—from cheap (rule-based, data-independent) to expensive

(agentic, data-dependent)—but also in expressiveness: e.g., agentic rewrites can synthesize novel transformations. MOAR is the only query optimizer with rewrites that span all four quadrants.

(2) **Cost model.** For semantic operators, the new challenge is estimating accuracy in addition to monetary cost or latency: there is no closed-form expression for how well an LLM will perform a task [26, 46]. Approaches to accuracy estimation vary in cost and precision. The simplest approach naively executes candidate plans on samples (as in LOTUS, DocETL-V1, and MOAR). ABACUS uses multi-armed bandits to adaptively sample plans and estimate their accuracy, tightening the accuracy bounds for promising (i.e., Pareto frontier) candidate plans. Some rewrites guarantee accuracy by construction, e.g., model cascades bound accuracy relative to the unrewritten plan [21, 22, 36, 58]. All approaches require some ground truth—either a user-specified “oracle” LLM-powered implementation [22, 36, 41], a user-defined accuracy function [41, 56], or LLM-as-judge evaluation approaches [44].

(3) **Search algorithm.** Given a plan space and cost model, a search algorithm finds good plans. Classical optimizers use dynamic programming, composing optimal subplans [16, 17]. Semantic operators pose two new challenges: plans may lack optimal substructure (Sec. 1), and adding a new objective, accuracy, requires returning a Pareto frontier of cost-accuracy tradeoffs for users. ABACUS adapts Cascades [16] to return such a frontier; DocETL-V1 uses top-down search with LLM-as-judge evaluation; and MOAR uses UCT-based search [25] over complete pipelines. ThalamusDB and LOTUS optimize parameters within individual rewrites (e.g., cascade thresholds) but not across rewrites, while other systems help users rewrite pipelines interactively [31, 45].

Overall, MOAR spans the broadest rewrite space of any semantic query optimizer, returns a frontier of low-cost, high-accuracy plans, and handles the lack of optimal substructure in query plans.

7 CONCLUSION

We introduce MOAR (Multi-Objective Agentic Rewrites), a novel optimizer for LLM-powered data processing pipelines that jointly optimizes for both accuracy and cost. Building on DocETL’s foundation of agentic rewrite directives, MOAR introduces (i) an expanded library of over 30 rewrite directives, (ii) a multi-armed bandit-based search algorithm that efficiently discovers sequences of rewrites that lead to good plans, and (iii) comprehensive empirical validation across six real-world workloads, demonstrating substantial improvements over state-of-the-art systems. MOAR achieves the highest accuracy on all workloads. Compared to ABACUS [41], the next-best optimizer, MOAR achieves 27% higher accuracy on average while matching its best accuracy at only 55% of its cost.

ACKNOWLEDGMENTS

We acknowledge support from grants DGE-2243822, IIS-2129008, IIS-1940759, and IIS-1940757 awarded by the NSF, funds from the State of California, funds from the Alfred P. Sloan Foundation, as well as EPIC lab sponsors: Adobe, Google, G-Research, Microsoft, PromptQL, Sigma Computing, Snowflake, Jane Street, and Bridgewater. Compute credits were provided by Azure, Modal, NSF (via NAIRR), and OpenAI. SS was supported by an NDSEG Fellowship.

REFERENCES

- [1] Asma Ben Abacha, Wen-wai Yim, Yajuan Fu, Zhaoyi Sun, Meliha Yetisgen, Fei Xia, and Thomas Lin. 2024. Medec: A benchmark for medical error detection and correction in clinical notes. *arXiv preprint arXiv:2412.19260* (2024).
- [2] Paritosh Aggarwal, Bowei Chen, Anupam Datta, Benjamin Han, Boxin Jiang, Nitish Jindal, Zihan Li, Aaron Lin, Pawel Liskowski, Jay Tayade, et al. 2025. Cortex AISQL: A Production SQL Engine for Unstructured Data. *arXiv preprint arXiv:2511.07663* (2025).
- [3] Aider Project. 2025. Edit Formats. <https://aider.chat/docs/more/edit-formats.html>. Describes the diff edit format used for LLM-based code editing via search/replace blocks.
- [4] Eric Anderson, Jonathan Fritz, Austin Lee, Bohou Li, Mark Lindblad, Henry Lindeman, Alex Meyer, Parth Parmar, Tanvi Ranade, Mehul A. Shah, Benjamin Sowell, Dan Tecuci, Vinayak Thapliyal, and Matt Welsh. 2025. The Design of an LLM-powered Unstructured Analytics System. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. <https://mail.vldb.org/cidrb/papers/2025/p13-anderson.pdf>
- [5] Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Ré. 2023. Language Models Enable Simple Systems for Generating Structured Views of Heterogeneous Data Lakes. *Proceedings of the VLDB Endowment* 17, 2 (2023), 92–105.
- [6] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47, 2 (2002), 235–256.
- [7] Oren Ben-Kiki, Clark Evans, and Ingy dot Net. 2009. YAML Ain't Markup Language (YAML™) Version 1.2. <https://yaml.org/spec/1.2/spec.html>. W3C YAML 1.2 Spec.
- [8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [9] John M. Carroll and Mary Beth Rosson. 1987. Paradox of the Active User. In *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*, John M. Carroll (Ed.). MIT Press, 80–111. Introduces the concept of progressive disclosure in user interface design.
- [10] Guillaume M Jb Chaslot, Mark HM Winands, H Jaap van den Herik, Jos WHM Uiterwijk, and Bruno Bouzy. 2008. Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation* 4, 03 (2008), 343–357.
- [11] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 34–43.
- [12] Samuel Colvin and the Pydantic contributors. 2025. *Pydantic: Data validation and settings management using Python type annotations*. <https://pypi.org/project/pydantic/>. Accessed: 2025-10-27.
- [13] Anas Dorbani, Sunny Yasser, Jimmy Lin, and Amine Mhedhbi. 2025. Beyond Quacking: Deep Integration of Language Models and RAG into DuckDB. *Proc. VLDB Endow.* 18, 12 (Sept. 2025), 5415–5418. <https://doi.org/10.14778/3750601.3750685>
- [14] Till Döhmen. 2024. Introducing the prompt() Function: Use the Power of LLMs with SQL! [urlhttps://motherduck.com/blog/sql-llm-prompt-function-gpt-models/](https://motherduck.com/blog/sql-llm-prompt-function-gpt-models/). Accessed: 2025-06-22.
- [15] Google Cloud. 2025. Perform intelligent SQL queries using AlloyDB AI query engine. <https://cloud.google.com/alloydb/docs/ai/evaluate-semantic-queries-ai-operators>. Accessed: 2025-06-22; Last updated: 2025-06-11.
- [16] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data(base) Engineering Bulletin* 18 (1995), 19–29. <https://api.semanticscholar.org/CorpusID:260706023>
- [17] Goetz Graefe and William J McKenna. 1993. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th international conference on data engineering*. IEEE, 209–218.
- [18] Jian He and Vaibhav Sethi. 2025. *SQL Reimagined for the AI Era with BigQuery AI Functions*. Google Cloud. <https://cloud.google.com/blog/products/data-analytics/sql-reimagined-for-the-ai-era-with-bigquery-ai-functions> Accessed: [insert date you accessed the article].
- [19] Joseph M. Hellerstein and Michael Stonebraker. 1993. Predicate migration: optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (Washington, D.C., USA) (SIGMOD '93). Association for Computing Machinery, New York, NY, USA, 267–276. <https://doi.org/10.1145/170035.170078>
- [20] Chuxuan Hu, Austin Peters, and Daniel Kang. 2024. LEAP: LLM-Powered End-to-End Automatic Library for Processing Social Science Queries on Unstructured Data. *Proceedings of the VLDB Endowment* 18, 2 (2024), 253–264.
- [21] Saehan Jo and Immanuel Trummer. 2024. Thalamusdb: Approximate query processing on multi-modal data. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.
- [22] Saehan Jo and Immanuel Trummer. 2025. SpareLLM: Automatically Selecting Task-Specific Minimum-Cost Large Language Models under Equivalence Constraint. *Proc. ACM Manag. Data* 3, 3, Article 219 (June 2025), 26 pages. <https://doi.org/10.1145/3725356>
- [23] Omar Khattab, Arnab Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, Heather Miller, et al. 2024. Dspy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*.
- [24] Kimi Team. 2025. Kimi K2: Open Agentic Intelligence. (2025). [arXiv:2507.20534](https://arxiv.org/abs/2507.20534)
- [25] Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*. Springer, 282–293.
- [26] Alexander W. Lee, Justin Chan, Michael Fu, Nicolas Kim, Akshay Mehta, Deepti Raghavan, and Ugur Cetintemel. 2025. Semantic Integrity Constraints: Declarative Guardrails for AI-Augmented Data Processing Systems. *Proc. VLDB Endow.* 18, 11 (Sept. 2025), 4073–4080. <https://doi.org/10.14778/3749646.3749677>
- [27] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.
- [28] Zequn Li, Yuanhao Zhong, Chengliang Chai, Zhaoze Sun, Yuhao Deng, Ye Yuan, Guoren Wang, and Lei Cao. 2025. DocDB: A Database for Unstructured Document Analysis. *Proc. VLDB Endow.* 18, 12 (Aug. 2025), 5387–5390. <https://doi.org/10.14778/3750601.3750678>
- [29] Yiming Lin, Madelon Hulsebos, Ruiying Ma, Shreya Shankar, Sepanta Zeighami, Aditya G Parameswaran, and Eugene Wu. 2025. Querying Templated Document Collections with Large Language Models. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2422–2435.
- [30] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, et al. 2025. Palimpsest: Optimizing ai-powered analytics with declarative query processing. In *Proceedings of the Conference on Innovative Database Research (CIDR)*. 2.
- [31] Chunwei Liu, Gerardo Vitagliano, Brandon Rose, Matthew Printz, David Andrew Samson, and Michael Cafarella. 2025. Palimpsest: Declarative and Interactive AI analytics. In *Companion of the 2025 International Conference on Management of Data*. 183–186.
- [32] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* 55, 9, Article 195 (Jan. 2023), 35 pages. <https://doi.org/10.1145/3560815>
- [33] Meta AI. 2024. The Llama 3 Herd of Models. (2024). [arXiv:2407.21783](https://arxiv.org/abs/2407.21783)
- [34] Meta AI. 2026. The Llama 4 Herd: Architecture, Training, Evaluation, and Deployment Notes. (2026). [arXiv:2601.11659](https://arxiv.org/abs/2601.11659)
- [35] OpenAI. [n.d.]. OpenAI MRCR: Long Context multiple-needle-in-a-haystack benchmark. <https://huggingface.co/datasets/openai/mrcr>. <https://huggingface.co/datasets/openai/mrcr>
- [36] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2025. Semantic Operators and Their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4171–4184.
- [37] Qwen Team. 2025. Qwen3 Technical Report. (2025). [arXiv:2505.09388](https://arxiv.org/abs/2505.09388)
- [38] Kiran Ramnath, Kang Zhou, Sheng Guan, Soumya Smruti Mishra, Xuan Qi, Zhengyuan Shen, Shuai Wang, Sangmin Woo, Sullam Jeoung, Yawei Wang, et al. 2025. A systematic survey of automatic prompt optimization techniques. *arXiv preprint arXiv:2502.16923* (2025).
- [39] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.
- [40] Armin Ronacher. 2008. Jinja2 Templating Engine. <https://jinja.palletsprojects.com/>. Version 3.x. Accessed 2025-11-21.
- [41] Matthew Russo, Sivaprasad Sudhir, Gerardo Vitagliano, Chunwei Liu, Tim Kraska, Samuel Madden, and Michael Cafarella. 2025. Abacus: A Cost-Based Optimizer for Semantic Operator Systems. *arXiv preprint arXiv:2505.14661* (2025).
- [42] Gabriele Sanmartino, Matthias Urban, Paolo Papotti, and Carsten Binnig. 2026. The Stretto Execution Engine for LLM-Augmented Data Systems. *arXiv preprint arXiv:2602.04430* (2026).
- [43] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. 23–34.
- [44] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G Parameswaran, and Eugene Wu. 2025. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. *Proceedings of the VLDB Endowment* 18, 9 (2025), 3035–3048.
- [45] Shreya Shankar, Bhavya Chopra, Mawil Hasan, Stephen Lee, Bjoern Hartmann, Joseph Hellerstein, Aditya Parameswaran, and Eugene Wu. 2025. Steering Semantic Data Processing With DocWrangler. In *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology (UIST '25)*. Association for Computing Machinery, New York, NY, USA, Article 84, 18 pages. <https://doi.org/10.1145/3746059.3747625>
- [46] Shreya Shankar, Haotian Li, Parth Asawa, Madelon Hulsebos, Yiming Lin, J. D. Zamfirescu-Pereira, Harrison Chase, Will Fu-Hinthorn, Aditya G. Parameswaran,

- and Eugene Wu. 2024. spade: Synthesizing Data Quality Assertions for Large Language Model Pipelines. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4173–4186. <https://doi.org/10.14778/3685800.3685835>
- [47] Shreya Shankar, Sepanta Zeighami, and Aditya G. Parameswaran. 2026. Task Cascades for Efficient Unstructured Data Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. https://www.shreya.com/task_cascades_preprint.pdf To appear.
- [48] Snowflake. 2025. Introducing Cortex AISQL: Reimagining SQL into AI Query Language for Multimodal Data. <https://www.snowflake.com/en/blog/ai-sql-query-language/>. Accessed: July 15, 2025.
- [49] Zhaoze Sun, Chengliang Chai, Qiyang Deng, Kaisen Jin, Xinyu Guo, Han Han, Ye Yuan, Guoren Wang, and Lei Cao. 2025. QUEST: Query Optimization in Unstructured Document Analysis. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4560–4573.
- [50] Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, et al. 2026. Kimi K2. 5: Visual Agentic Intelligence. *arXiv preprint arXiv:2602.02276* (2026).
- [51] 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*. 472–475.
- [52] Matthias Urban and Carsten Binnig. 2024. Eleet: Efficient learned query execution over text and tables. *Proceedings of the VLDB Endowment* 17, 13 (2024), 4867–4880.
- [53] Kiran Vodrahalli, Santiago Ontañón, Nilesh Tripuraneni, Kelvin Xu, Sanil Jain, Rakesh Shivanna, Jeffrey Hui, Nishanth Dikkala, Mehran Kazemi, Bahare Fatemi, Rohan Anil, Ethan Dyer, Siamak Shakeri, Roopali Vij, Harsh Mehta, Vinay Ramasesh, Quoc Le, Ed Chi, Yifeng Lu, Orhan Firat, Angeliki Lazaridou, Jean-Baptiste Lespiau, Nithya Attaluri, and Kate Olszewska. 2024. Michelangelo: Long Context Evaluations Beyond Haystacks via Latent Structure Queries. *arXiv preprint arXiv:2409.12640* (2024). <https://arxiv.org/abs/2409.12640>
- [54] Jiayi Wang and Guoliang Li. 2025. Aop: Automated and interactive llm pipeline orchestration for answering complex queries. CIDR.
- [55] Jiayi Wang, Yuan Li, Jianming Wu, Shihui Xu, and Guoliang Li. 2025. Unify: A System For Unstructured Data Analytics. *Proc. VLDB Endow.* 18, 12 (Aug. 2025), 5287–5290. <https://doi.org/10.14778/3750601.3750653>
- [56] Lindsey Linxi Wei, Shreya Shankar, Sepanta Zeighami, Yeounoh Chung, Fatma Ozcan, and Aditya G Parameswaran. 2025. Multi-Objective Agentic Rewrites for Unstructured Data Processing. *arXiv preprint arXiv:2512.02289* (2025).
- [57] Patrick Wendell, Eric Peter, Nicolas Pelaez, Jianwei Xie, Vinny Vijeyakumaar, Linhong Liu, and Shitao Li. 2023. Introducing AI Functions: Integrating Large Language Models with Databricks SQL. <https://www.databricks.com/blog/2023/04/18/introducing-ai-functions-integrating-large-language-models-databricks-sql.html>. Accessed: 2025-06-22.
- [58] Sepanta Zeighami, Shreya Shankar, and Aditya Parameswaran. 2025. Cut Costs, Not Accuracy: LLM-Powered Data Processing with Guarantees. *arXiv preprint arXiv:2509.02896* (2025).
- [59] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems* 36 (2023), 46595–46623.
- [60] Eckart Zitzler. 1999. *Evolutionary algorithms for multiobjective optimization: Methods and applications*. Vol. 63. Shaker Ithaca.