Multi-Objective Agentic Rewrites for Unstructured Data Processing

Lindsey Linxi Wei[1†], Shreya Shankar[2†], Sepanta Zeighami[2],
Yeounoh Chung[3], Fatma Ozcan[3], Aditya G. Parameswaran[2]
[1]University of Washington, [2]UC Berkeley, [3]Systems 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 November 2025, has 3.2K GitHub stars and users across domains (e.g., journalism, law, medicine, policy, finance, and urban planning). In DocETL, users build pipelines by composing operators described in natural language, also known as semantic operators, with an LLM executing each operator's logic. However, due to complexity in the operator or the data it operates on, LLMs often give inaccurate results. To address this challenge, DocETL introduced *rewrite directives*, or abstract rules that guide LLM agents in rewriting pipelines by decomposing operators or data. For example, decomposing a single `filter("is this email sent from an executive and discussing fraud?")` into the conjunction of two separate semantic filters may improve accuracy. However, DocETL only optimizes for accuracy, not cost. *How do we optimize for both?*

We present MOAR (**M**ulti-**O**bjective **A**gentic **R**ewrites), a new optimizer for DocETL. To target cost optimization, we introduce two new categories of directives and extend all three existing categories with new ones, bringing the total to over 30 directives—more than doubling what DocETL originally had. Moreover, since operators can interact with each other unpredictably due to LLM behavior, optimizing operators or sub-pipelines individually can yield suboptimal overall plans. Recognizing this, we design a new global search algorithm that explores rewrites in the context of entire pipelines. Since the space of rewrites is infinite—pipelines can be rewritten in many ways, and each rewritten pipeline can itself be rewritten—our algorithm adapts 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**.

1 INTRODUCTION

LLMs are now integrated into systems that support queries over unstructured data, from both industry and academia [20, 24, 33, 38, 44, 48, 57]. In these LLM-powered data processing systems, a query is expressed as a sequence 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:

System	Multi-Objective	Rewrite Coverage	Global Search
LOTUS [38]	✗	✗	No search
ABACUS [42]	✓	✗	✗
DocETL-V1 [44]	✗	✓	✗
DocETL–MOAR	✓	✓✓	✓

Table 1: Comparison of semantic operator system query optimizers. MOAR (ours) is multi-objective, covers a broad space of rewrites, and searches without assuming optimal substructure.

Example 1.1 (Enhancement factors in public defender workloads). Public defenders that we work with 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 typically rely on the principle of *optimal substructure*: an optimal plan can be constructed from optimal solutions to its subplans.[1] For example, the Cascades framework [18] 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***

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

[1]Even 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.

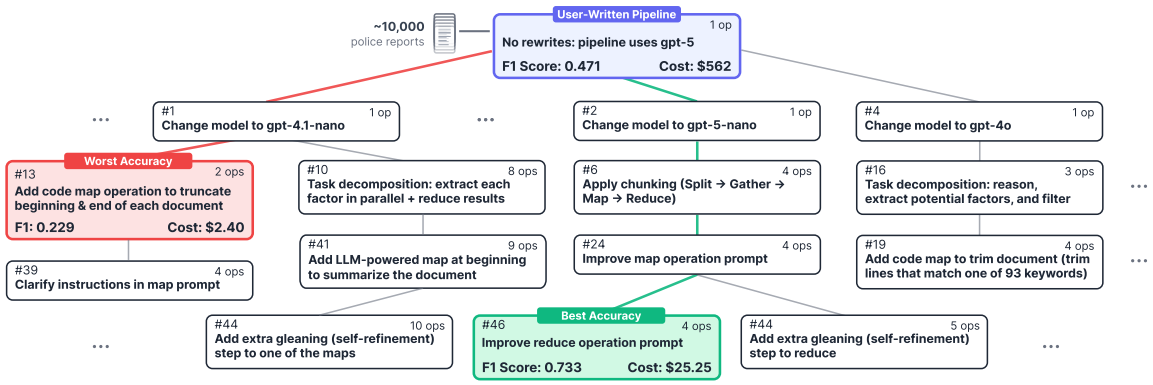


Figure 1: Sample of pipelines explored by MOAR when optimizing the pipeline in Ex. 1.1. The user-written pipeline (top, purple) contains a single `map` operator that extracts all enhancement factors. Each node is a pipeline variant produced by a rewrite (e.g., model change, code synthesis, task decomposition, 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.

unpredictable ways, composing locally optimal subplans can yield globally suboptimal plans.

Existing optimizers for LLM-powered data processing have limitations as summarized in Table 1. LOTUS [38] 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 semantic operator. ABACUS [42] performs cost-based search using transformation rules (i.e., logical-to-logical) from traditional data processing (e.g., filter pushdown) and 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 framework [18], which relies on optimal substructure to search for Pareto-optimal query plans. Our original DocETL optimizer [44] (that we call DocETL-V1) relaxes the optimal substructure assumption slightly in its search algorithm, but still optimizes operators from upstream to downstream, potentially missing beneficial rewrites where changing downstream operators may benefit from different upstream choices. Moreover, DocETL-V1 optimizes only for accuracy, not cost.

The MOAR optimizer. In this paper, we introduce the ***Multi-Objective Agentic Rewrites (MOAR)*** optimizer, which surfaces a Pareto frontier of pipelines that improve accuracy over the user's original while offering a range of costs—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 new directives to DocETL, bringing the total to over 30—capturing rewrites that are informed by our real deployments but have not been systematically explored in prior work.

Then, during the rewrite process, we observed that when the LLM agent receives the full specifications of all 30+ directives simultaneously in its prompt, it struggles to select appropriate rewrites. Inspired by *progressive disclosure* from HCI [9]—a technique that reduces cognitive load in user interfaces by revealing information gradually—we structure the agent's interaction with directives in stages. Agents initially see only directive names and high-level descriptions. When an agent selects a directive, it loads the full specification (detailed descriptions, instantiation schemas, examples) on-demand. Additionally, the agent can invoke a `read_document` tool at any point during instantiation to inspect sample data, grounding its decisions in actual document characteristics.

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. 3)—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 [42]), 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 3 summarizes all notation.

2.1 Datasets, Operators, and Pipelines

Datasets. A *dataset* D is a collection of documents, 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. [38]. 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 [41]; *(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 $\texttt{map}_x$, $\texttt{reduce}_x$, and so on. Unless otherwise noted, o_x (and typed forms like $\texttt{map}_x$, $\texttt{reduce}_x$) 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 $\texttt{code\_map}_x$, $\texttt{code\_reduce}_x$, 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 $\texttt{map}_x$ 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 $\texttt{map}_x(D)$ 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 $(o^{(1)}, \ldots, o^{(k)})$; given an input dataset D, we write its execution as $o^{(1)} \to o^{(2)} \to \cdots \to o^{(k)}$, where $\to$ denotes function composition, following Shankar et al. [44]: $o^{(k)}\big(o^{(k-1)}(\cdots o^{(1)}(D)\cdots)\big)$. 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 operators (split, gather, unnest). The auxiliary operators do not invoke LLMs. 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 Rewrites and Directives

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

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 $\texttt{map}_x$ and RHS $\texttt{code\_split} \rightarrow \texttt{code\_gather} \rightarrow \texttt{map}_{x'} \rightarrow \texttt{reduce}$, 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 $\texttt{map}_{x'}$ and the new reduce operator. To illustrate how directives combine to transform a pipeline, Fig. 2 shows the sequence of rewrites that leads from the user-authored pipeline to the highest-accuracy pipeline in Fig. 1.

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 $c(\texttt{code\_op}) = 0$, since we want to minimize monetary cost. The cost of a pipeline $P = (o_1, \ldots, o_k)$ 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. 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.

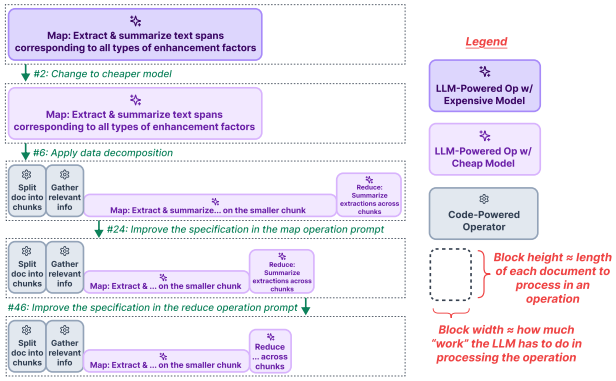


Figure 2: Illustration of the rewrite sequence that leads to the highest-accuracy pipeline in Fig. 1, qualitatively showing how rewrites restructure the pipeline so that LLMs perform less work on smaller portions of each document.

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

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.

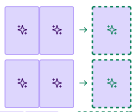
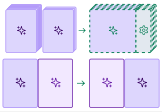
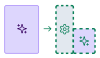
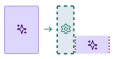
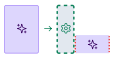
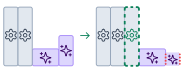
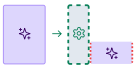
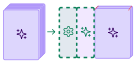
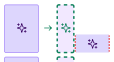
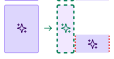
Objective. Let $\mathcal{P}$ denote the set of pipelines reachable from the initial user-authored pipeline P_0 through any sequence of rewrites. $\mathcal{P}$ is clearly infinite, as there are an arbitrary number of ways to instantiate a given rewrite directive. Let B denote the evaluation budget, i.e., the maximum number of candidate pipelines that can be executed and scored. For simplicity, we define the budget in terms of the number of pipeline evaluations, though it could be extended to wall-clock time or total monetary cost. The optimizer selects a subset $\mathcal{P}_B \subseteq \mathcal{P}$ with $|\mathcal{P}_B| \le B$ and returns the approximate Pareto frontier $\hat{\mathcal{F}} = \text{Pareto}(\mathcal{P}_B)$. The optimization objective is then:

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

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. 3): an expanded library of rewrite directives (Sec. 3) and a search algorithm that efficiently discovers high-quality pipelines within the evaluation budget (Sec. 4).

Table 2: Summary of new rewrite directives in MOAR. Categories marked with † are novel to MOAR. Directives marked with ‡ generate multiple candidate pipelines and select the highest-accuracy pipeline after evaluation on D_o. The "visual" column qualitatively depicts each rewrite (as in Fig. 2): 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	$\texttt{map}_x \rightarrow \texttt{map}_y \Rightarrow \texttt{map}_z$; similarly for `filter` and `reduce`	Fuses pairs of same-type operators (`map`–`map`, `filter`–`filter`, `reduce`–`reduce`) into a single operator.	
	❷ Map–Reduce	$\texttt{map}_x \rightarrow \texttt{reduce}_{K,y} \Rightarrow \texttt{reduce}_{K,z}$	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	$\texttt{map}_x \rightarrow \texttt{filter}_y \Rightarrow \texttt{map}_z \rightarrow \texttt{code\_filter}$	Expands the `map` to also compute the predicate produced by the downstream operator $\texttt{filter}_y$ (so z's output schema is the union of those from x and y), followed by a `code_filter` that simply checks the boolean attribute generated (in y's schema).	
	❹ Filter–Map	$\texttt{filter}_x \rightarrow \texttt{map}_y \Rightarrow \texttt{map}_z \rightarrow \texttt{code\_filter}$	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	$o_x \Rightarrow \texttt{code\_op}_{\hat{x}}$	Replaces an LLM-powered operator with synthesized Python code.	
	❼ Code Sub. (Reduce)	$\texttt{reduce}_x \Rightarrow \texttt{code\_reduce}_{\hat{x}} \rightarrow \texttt{map}$	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_reduce` counts themes and concatenates relevant context, and the `map` generates the natural-language report.	
	❽ Doc. Compression (Code)‡	$o_x \Rightarrow \texttt{code\_map} \rightarrow o_{x'}$	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 Compr.‡	$o_x \Rightarrow \texttt{code\_map} \rightarrow o_{x'}$	Retains only the first h and last ℓ words (or lines) of each document via a synthesized `code_map`. Useful when key information typically appears at document boundaries (e.g., abstract, conclusion).	
Data Decomposition	❿ Chunk Sampling‡	$\texttt{split} \rightarrow \texttt{gather} \rightarrow \texttt{map} \rightarrow \texttt{reduce} \Rightarrow \texttt{split} \rightarrow \texttt{gather} \rightarrow \texttt{sample} \rightarrow \texttt{map} \rightarrow \texttt{reduce}$	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‡	$\texttt{reduce}_{K,x} \Rightarrow \texttt{sample}_K \rightarrow \texttt{reduce}_{K,x}$	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‡	$\texttt{filter}_x \Rightarrow \texttt{code\_filter}^* \rightarrow \texttt{filter}^*_y \rightarrow \texttt{filter}_x$	Inserts one or two cheaper "pre-filters" before $\texttt{filter}_x$. `code_filter` and $\texttt{filter}_y$ (marked with * to indicate optionality; at least one is required) form a cascade in which each pre-filter removes additional documents before they reach $\texttt{filter}_x$.	
Projection Synthesis	⓭ Doc. Summarization	$o_x \Rightarrow \texttt{map} \rightarrow o_{x'}$	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)‡	$o_x \Rightarrow \texttt{extract} \rightarrow o_{x'}$	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.	
LLM-Centric	⓯ Model Substitution	$o_x \Rightarrow o_{x'}$ where $x' = (p, s, m')$	Replaces an operator's LLM with a different model.	
	⓰ Clarify Instructions‡	$o_x \Rightarrow o_{x'}$ where $x' = (p', s, m)$	Rewrites the prompt template to be more specific and detailed, reducing ambiguity and thus making the task "easier" for the LLM.	
	⓱ Few-shot Examples	$o_x \Rightarrow o_{x'}$ where $x' = (p', s, m)$	Adds few-shot examples to prompts (a standard strategy for improving accuracy [8]), thus making the task "easier" for the LLM.	
	⓲ Arbitrary Rewrite†	$P \Rightarrow P'$	Allows the agent to propose free-form pipeline transformations beyond the predefined directives.	

3 NEW OPERATORS AND DIRECTIVES

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* (combining or rearranging operators) and *code synthesis* (replacing or adding code-powered operations). Second, we adapt existing 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. Together with DocETL-V1's 13 directives,

MOAR's 18 new directives bring the total to over 30. Table 2 summarizes the new directives; detailed descriptions are in App. B. We highlight a few examples below (directive numbers reference the corresponding row in Table 2):

- In ***Fusion and Reordering.*** The 3 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. (4 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 5 so the `filter` immediately follows the firearm-extraction `map`, and fuse them—avoiding downstream processing for irrelevant factors.
- In ***Code Synthesis.*** The 8 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 6 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 [40], 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. For instance, when extracting enhancement factors from police reports (Ex. 1.1), if a prior `split` operator has divided each report into chunks—each chunk now becoming a document—then `sample` can issue a BM25 query with terms like "firearm" and "weapon" or use embedding similarity to a query such as "threatening with a weapon" to select only the relevant chunks before the downstream `map`. By processing fewer documents, the subsequent LLM-powered operator incurs lower cost. 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.

Table 3: 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'
	d	A rewrite directive
Cost & Accuracy	$c(P)$; $a(P)$; $\hat{c}(P)$; $\hat{a}(P)$	Cost and accuracy of P; empirical estimates on D_o
	B	Evaluation budget
Search Tree	$T_t = (V_t, E_t)$	Search tree with vertices V_t, edges E_t
	V_t; $\hat{\mathcal{F}}_t$; $\mathcal{F}$	Evaluated pipelines; frontiers
	children(P); parent(P)	Child/parent nodes
	desc$_t(P)$; depth(P)	Descendants; depth
	$last\_action(P)$	Directive used to generate P from parent(P)
Pipeline Selection & Utility	$A_t(P, c)$	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
	$W(n_t(P))$	Max children P may generate
Rewriting	$\nu(P, d)$	Usage count of d from P
	k	Number of candidate pipelines generated for a rewrite
	rank$_t(P)$	Accuracy rank of P compared to all pipelines explored

4 SEARCH ALGORITHM

In this section, we describe how MOAR efficiently searches over rewrite directives to discover high-quality and low-cost pipelines. Fig. 3 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 [18, 43], MOAR performs global search over complete pipelines. This design reflects 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. 3). 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

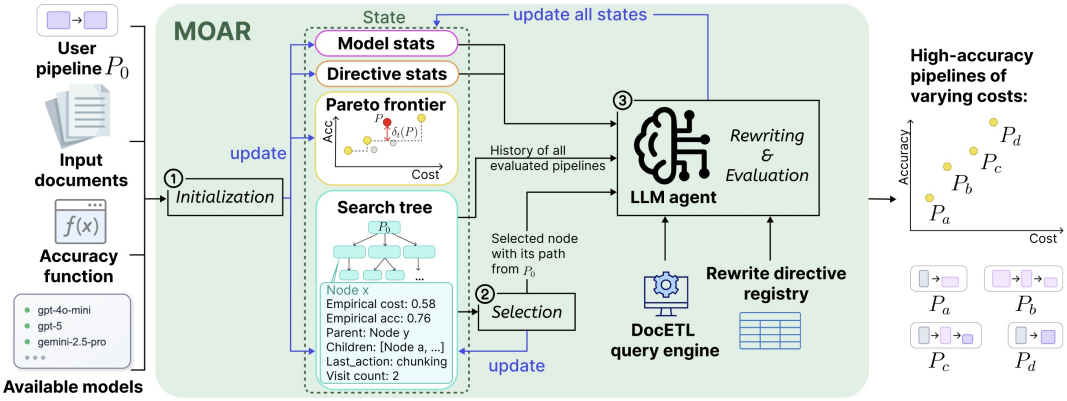


Figure 3: System architecture of MOAR. The optimizer takes as input a user pipeline P_0, input documents, user-authored accuracy function, and available models. In the initialization phase (①), MOAR evaluates P_0 with all models to construct an initial frontier. The main search loop alternates between 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.

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 $\hat{\mathcal{F}}_B$

Function MOAR(P_0, D_o, M, B):
  // Initialization: evaluate P_0 with all models
  $T \leftarrow$ Initialize(P_0, D_o, M);
  initialize directive usage map $\nu(P, d) \leftarrow 0$ for all P, d;
  $t \leftarrow |V|$; // Number of pipelines evaluated so far
  // Main search loop
  while $t < B$ **do**
    // Selection: traverse tree to find node to rewrite
    $P^\star \leftarrow$ Select(P_0, T_t);
    // Rewriting and evaluation (parallelized)
    $(P', r, \hat{c}(P'), \hat{a}(P'), k) \leftarrow$ RewriteAndEvaluate($P^\star, G_t, \nu, D_o$);
    // Update tree with new pipeline and rewrite edge
    add P' to V and edge $P^\star \xrightarrow{r} P'$ to E;
    $t \leftarrow t + k$; // Increment by number of candidates evaluated
    recompute $\hat{\mathcal{F}}_t$ from V;
  end
  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. 3. The search process begins with *initialization*, where MOAR constructs an initial frontier 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, guided by how much the pipeline and its descendants contribute to the Pareto frontier. Second, in the *rewriting and evaluation phase*, an 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 3 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).[2] 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. 3), 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) parent(P) and 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) *last_action*(P): the directive applied to generate P from parent(P).

Most statistics remain fixed throughout search; only 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

[2] If $|M| > C_m$ (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.

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 $P^\star$ 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

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

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. 3.

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 [28], 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 $P^\star$: starting at P_0, we repeatedly select the child with the highest utility until reaching a node $P^\star$ that can generate a new child. In standard UCT [28], 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+ directives × multiple operators × 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 specifies the complete procedure.

4.3 Rewriting and Evaluation

Given the selected pipeline $P^\star$, 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 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. Our implementation uses 3 workers (capped by LLM API rate limits). 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` ⇒ `code_reduce` → `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.
- **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.
- **Test cases.** Scenarios specifying an input pipeline, target operators, expected transformation behavior, and whether the test should pass. These tests ensure that LLM agents can understand the directive specifications and instantiate rewrites correctly.

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 $P^\star$. 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.[3]

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

- Pipeline $P^\star$ (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 $P^\star$ and its depth($P^\star$).
- Model and directive statistics, as defined in Sec. 4.1.
- An objective—either "reduce cost while preserving accuracy" or "improve accuracy"—determined by $P^\star$'s rank among evaluated pipelines. If $\text{rank}_t(P^\star) \leq |V_t|/2$ (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 a directive d whose LHS matches a subsequence of operators in $P^\star$ 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 $P^\star \xrightarrow{r} P'$. 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 $P^\star$ 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 (3 in our implementation). 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. Such directives are marked with ‡ in Table 2 and App. B.

4.3.3 Evaluation and Error Handling Once a valid rewrite P' is generated, MOAR executes it on the evaluation sample D_o to obtain empirical cost and accuracy $(\hat{c}(P'), \hat{a}(P'))$. If an identical pipeline was evaluated previously, MOAR reuses the cached measurements. The measured statistics are recorded in the tree along with the pipeline's depth and parent pointer.

Any agentic query optimizer must handle errors that arise when delegating decisions to LLM agents and evaluating generated pipelines. MOAR encounters three types of errors. First, the agent may choose a directive whose applicability signature does not match $P^\star$, or generate a rewrite that cannot be parsed by the DocETL query engine; in both cases, MOAR retries by invoking the agent again. Second, LLM API errors may occur during pipeline execution on D_o (e.g., rate limits, service outages); MOAR discards these pipelines without retry since they indicate transient infrastructure issues rather than problems with the rewrite. If a retry also fails, the pipeline is discarded. When any pipeline is discarded, MOAR decrements the visit count of $P^\star$ to ensure that failed attempts do not artificially inflate visit counts.

Once the evaluation budget B has been exhausted, MOAR outputs the final Pareto frontier $\hat{\mathcal{F}}_B$ constructed from all evaluated pipelines.

5 EVALUATION

We evaluate MOAR across a diverse set of workloads spanning legal, biomedical, government, consumer, and corporate domains. ***MOAR discovers the highest-accuracy pipeline on every workload.*** On average, MOAR achieves 27% higher accuracy than the next-best optimizer, ABACUS [42], 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.

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 [38, 42, 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.

[3] Claude Code uses a similar approach for presenting documentation to agents [59].

5.1.1 Baselines We compare MOAR against 4 baseline systems, including the original DocETL-V1 optimizer and a naive "agentic" baseline. We compare against open-source systems that support a semantic `map` operator. For each baseline, we express the pipeline using the baseline's interface, minimally modifying the DocETL operators so the pipeline can be parsed by the baseline.

- **DocETL-V1 [44].** We run the original DocETL optimizer, which optimizes only for accuracy. DocETL-V1 returns a single plan.
- **Simple agent.** We test whether an LLM agent can discover effective rewrites for DocETL, *without* explicit directives or structured search provided by MOAR. We provide a `gpt-5` agent with three tools: (i) reading sample documents, (ii) reading DocETL documentation or the original DocETL paper [44], and (iii) executing any pipeline on samples and observing its accuracy and cost. The agent proposes rewrites until its context window is exhausted (≈400k tokens) or it calls a "done" tool to indicate completion. Then, of *all* pipelines generated by the agent, we retain the Pareto frontier.
- **LOTUS [38].** 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.
- **Palimpzest/ABACUS [33, 42].** We express each pipeline using Palimpzest (PZ)'s [33] operators and use its ABACUS optimizer [42]. 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.

PZ and LOTUS each accept a `num_samples` parameter for optimization (PZ defaults to 100, LOTUS to 200); we set both to 200 to provide each system ample budget for exploration (unless mentioned otherwise in Sec. 5.1.2). We report optimization overheads for all methods in App. D, Table 9.

5.1.2 Workloads We evaluate across six workloads. We take text processing workloads identified in prior published work [38, 42, 44], and introduce two new workloads from medical analysis and enterprise sustainability domains, inspired by real DocETL users. The Sustainability workload is inspired by the Scottish Climate Intelligence Service, who uses DocETL to identify common sustainability initiatives across organizations and regional authorities. The MEDEC workload is inspired by AnkiHub, who builds study tools for medical students (e.g., personalized quizzes and automated error detection in their reasoning). For privacy, we use public datasets with similar task structures rather than proprietary user data.

For each workload, we describe the task, initial pipeline, and metric. 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 sampled from larger source datasets, split into D_o (40 documents per workload, for optimization) and a held-out test set $D_T = D \setminus D_o$ (100 documents per workload). Optimizers search over candidate pipelines using D_o, and all reported results reflect execution on the held-out test set D_T.

- **CUAD (Legal Analysis) [22].** This dataset contains 510 legal contracts (each averaging 7,727 words) annotated with 41 clause categories, used as a benchmark in prior work [42, 44]. The task is to extract text spans for each clause type present in a contract. The initial pipeline consists of a single `map` that prompts for all clause types at once and outputs a list of {`clause_type`, `text_span`} objects. For PZ, we use the pipeline provided by the ABACUS authors, doubling the number of samples (as mentioned in Sec. 5.1.1) to aid PZ's exploration. The metric is F1 score, counting an extraction as correct if the clause type matches and the span has Jaccard similarity > 0.15 with the ground truth.
- **Game Reviews.** This dataset, from prior work [44], involves documents from the Steam reviews dataset [49], where each document contains 300 reviews for a single game (averaging 97,696 words). The task is to identify ten positive and ten negative reviews per game, presented in chronological order. The initial pipeline, as in [44], is a single `map` over each document that attempts to extract and order the required reviews directly. The metric is an average of hallucination rate (fraction of extracted reviews not present in the source text), sentiment accuracy (agreement with ground-truth ratings for non-hallucinated reviews), and Kendall's τ between predicted and correct orderings. Since documents exceed context windows of most LLMs, we use only `gpt-4.1` and `gemini` series models, with `gpt-4.1-mini` as default. We note that PZ crashed during optimization with a 200-sample budget (~4 hours in), so we reduced PZ's budget to 50 samples.
- **BlackVault (Declassified Articles).** This dataset from [44] contains 733 articles (each averaging 7,351 words) describing international paranormal events. The task is to classify each document's event type and aggregate distinct location mentions across documents of the same type. The initial pipeline, as in [44], has two operators: a `map` that extracts an event type per article (e.g., UFO sighting), followed by a `reduce` that aggregates locations by type. The metric is average recall of distinct locations per event type, normalized by the maximum recall achieved across all methods in Sec. 5.1.1 (to get a score between 0 and 1).
- **Biodex (Biomedical Classification).** This benchmark [16] evaluates biomedical paper classification (each averaging 71,151 words). Each paper must be linked to the adverse drug reactions it discusses, drawn from over 24,000. Prior work expressed this task using different pipelines (code provided by the respective paper authors). LOTUS-r&r (retrieve-and-rerank) implements a join (to find all reactions for each document) followed by a semantic aggregation (to group by document and rerank reactions with an LLM) [38]. PZ-r&r implements a map, retrieve, and map pipeline [42]. However, now that LLMs support sufficiently long context windows (all models in our pool exceed 128k tokens), we implement the simplest initial pipeline for MOAR: a single `map` operation where the prompt contains the full list of reactions and the output is a ranked list of reactions relevant to the document. For fair comparison, we also express this single-map initial pipeline in LOTUS and PZ, called LOTUS-d and PZ-d (direct). We report results for both pipeline formulations (d and r&r) for LOTUS and PZ, taking the union of plans

Table 4: Best accuracy by method. 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).

Workload	DocETL-V1	SA	LOTUS	PZ	MOAR
CUAD	0.471	0.521	0.402	<u>0.694</u>	**0.762**
Game Reviews	0.504	0.467	0.522	<u>0.683</u>	**0.804**
BlackVault	0.143	<u>0.194</u>	0.081	–	**1.000**
Biodex	0.247	<u>0.333</u>	0.260 (d) 0.202 (r&r)	0.260 (d) 0.296 (r&r)	**0.370**
Medec	0.534	<u>0.726</u>	0.538	0.536	**0.742**
Sustainability	<u>0.632</u>	0.543	0.516	–	**0.646**
Average Gain (%)	+135.26%	+94.36%	+209.53%	+26.65%	–

found for their Pareto frontiers. The metric is rank-precision@5 (RP@5), measuring how often the correct reactions appear among the top-5 predictions.

- **MEDEC (Medical Error Detection and Correction); new.** The MEDEC dataset [1] contains 3,848 clinical notes (each averaging 147 words) with labeled medical errors across five categories. The task is to detect whether an error is present in each note, identify the sentence containing the error, and generate a corrected version. The initial pipeline is a single `map` that produces three outputs: `error_flag`, `error_sentence`, and `corrected_sentence`. The metric is an average of the error-detection F1 scores and the corrected sentence Jaccard similarities with the reference corrections.
- **Sustainability (Corporate ESG Reports); new.** This dataset [13] contains 5,436 enterprise reports (each averaging 38,668 words) spanning annual reports, sustainability reports, financial reports, and others. The task is to (i) filter to retain only sustainability reports, (ii) classify each company's economic sector (e.g., health, real estate), and (iii) for each sector, produce a summary listing each company and its key sustainability initiatives (e.g., for the technology sector: "Apple: carbon neutrality by 2030; Microsoft: 100% renewable energy; Google: water replenishment programs"). The initial pipeline applies a `filter` to select sustainability reports, a `map` to classify the economic sector, and a `reduce` to group reports by sector and generate a summary per sector containing all companies and their initiatives. The metric is the average of sector classification accuracy (the fraction of reports assigned to the correct sector) and company name accuracy (the fraction of company names in the per-sector summaries that match ground-truth companies for that sector).

5.2 Results

We present results on accuracy (Sec. 5.2.1), cost savings and Pareto frontier quality (Sec. 5.2.2), and optimization overhead (Sec. 5.2.3).

5.2.1 Accuracy Improvement Table 4 summarizes MOAR's accuracy improvements across all baselines. ***MOAR achieves the highest accuracy on every workload, with gains over the next most accurate baseline ranging from +2.2% (Medec, Sustainability) to +415.5% (BlackVault).*** The smallest accuracy improvement is on Medec (+2.2% over the Simple Agent), where the Simple Agent's top-accuracy pipeline simply replaced the default model with `gpt-5`. One possible explanation is that documents are very short (147 words on average), and we use the optimized prompt from the original paper [1]—leaving minimal room for further optimization.

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.***

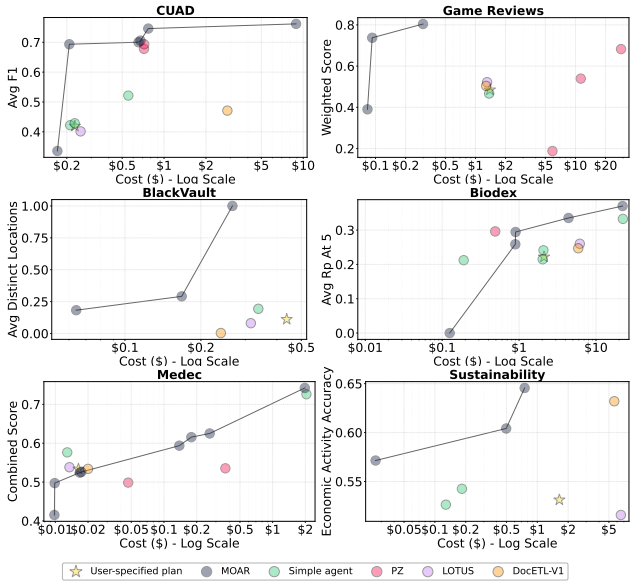


Figure 4: Pareto frontiers for each method, for each held-out test set. "User-specified plan" is the initial pipeline authored by the user.

5.2.2 Cost Savings and Pareto Frontier Quality Table 5 shows MOAR's cost to match each baseline's highest accuracy. App. D 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. 4 shows the accuracy-cost Pareto frontier for each method. MOAR completely dominates all other methods 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 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

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	SA	LOTUS	PZ
CUAD	0.073×	0.377×	–	0.290×
Game Reviews	0.072×	–	0.071×	0.003×
BlackVault	0.267×	0.497×	–	n/a
Biodex	0.152×	0.196×	0.145×	1.840×
Medec	0.840×	0.966×	1.245×	0.046×
Sustainability	0.133×	0.143×	–	n/a

Table 6: Model usage across 29 top-accuracy pipelines. Task types: Ext. = extraction, Class. = classification, Summ. = summarization.

	Task Type			Doc Length		
Model	Ext.	Class.	Summ.	Short	Long	**Frac.**
gpt-5-nano	✓	✓		✓	✓	41%
gemini-2.5-flash-lite		✓	✓		✓	17%
gpt-4.1-mini	✓			✓		14%
gpt-4o-mini			✓		✓	10%
gpt-5		✓		✓		10%

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. Prior work has shown that users are often willing to incur substantial optimization costs to achieve highest accuracies [27]. 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]. We report optimization costs and latencies for all methods in App. D, Table 9.

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).

- **Workloads exhibit steep accuracy-cost trade-offs.** The second-highest accuracy pipeline for MOAR was, on average, 18.81% less accurate but 66.36% cheaper. For CUAD, the second-best pipeline reduced cost by 91.34% with only a 2.07% accuracy drop. Game Reviews and Biodex showed similar trends, with 68.98% and 80.19% cost reductions for 8.30% and 9.41% decreases in accuracy respectively. These results underscore the importance of returning a Pareto frontier of high-accuracy pipelines so users can select which pipeline to use given their accuracy and cost constraints. Future work could explore how to discover user preferences or develop heuristics to guide users toward the region of the frontier that mest matches their needs.
- **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.
- **79% use projection synthesis.** These strategies reduce document size before LLM operations, which not only reduces cost but also helps LLMs focus on relevant information. Among them, 55% use deterministic methods (regex, full-text search, or code-based pruning), 17% use embedding-based pruning, and 14% use LLM-powered summarization.
- **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%.
- **Optimal models are workload-dependent.** All 29 pipelines switch from the default `gpt-4o-mini`, even though it is OpenAI's "most cost-efficient small model" [37]. Each workload's most accurate pipeline uses a different model. The model usage patterns reveal specializations (Table 6): `gpt-5-nano` is the most prevalent (41%), used for extraction and classification on both long and short documents, while smaller models like `gemini-2.5-flash-lite` and `gpt-4o-mini` are used for summarization on long documents.
- **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 30—demonstrating that MOAR avoids premature convergence and maintains strong exploration throughout.

	Rule-based	Agentic
Data-indep.	e.g., model substitution, ensembling, operator reordering ABACUS [42], FLOCK-MTL [14], UNIFY [56], DOCDB [31], MOAR	e.g., operator fusion, task decomposition DOCETL-V1 [44], CAESURA [51], UNIFY [56], AOP [55], MOAR
Data-dep.	e.g., model cascades, fine-tuning, RAG ABACUS [42], LOTUS [38], THALAMUSDB [24], CORTEX-AISQL [2], ELEET [52], UNIFY [56], DOCDB [31], MOAR	e.g., code synthesis, other rewrite directives DOCETL-V1 [44], CORTEX-AISQL [2], MOAR

Figure 5: 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.

6 RELATED WORK

We situate MOAR in the context of semantic data processing systems and semantic query optimization.

Semantic Data Processing Systems. Systems that expose AI-powered data processing capabilities have proliferated over the past two years. ThalamusDB [24] was among the earliest to support natural language filters and joins in SQL. LOTUS [38] coined the term "semantic operators" for this type of LLM-powered data processing; other systems that support at least one semantic operator include Palimpzest [33], Aryn [4], DocETL [44], FlockMTL [14], ELEET [52], Unify [56], and DocDB [31]. Other systems target specific applications like templatized documents [32], extraction tasks [5, 50], and social science queries [23]. Industrial systems from Databricks [57], DuckDB [15], Snowflake [2, 48], Google BigQuery [20], and Google AlloyDB [17] also now support LLM-powered functions. In our evaluation (Sec. 5), we test active open-source systems that support at least a semantic `map` operator—LOTUS, Palimpzest, and DocETL-V1.

Semantic Query Optimization. Query optimizers have three components: a plan space, a cost model, and a search algorithm. We compare semantic data processing optimizers for each.

(1) **Plan space.** Classical relational database optimizers search over rewrites such as filter and join reordering [11, 21, 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. 5): 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). Examples of rule-based, data-independent rewrites include physical implementations like model substitution and ensembling in ABACUS [42]. Examples of rule-based, data-dependent rewrites—those that require samples to learn configurations but not an LLM agent to instantiate—include model cascades [26, 38, 53, 58], context reduction [42], and fine-tuning small LLMs [52]. An example of a rewrite that requires an LLM agent to instantiate, but not sample data, is operator fusion. Examples of agentic, data-dependent rewrites include most MOAR rewrite directives. 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.** A cost model estimates the quality of candidate plans, guiding the search algorithm toward good plans. Selinger's cost model, for example, estimates I/O and CPU costs using statistics like cardinality and index availability [43]. 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 [29, 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; this cost estimation technique could be integrated into MOAR's search algorithm. Some rewrites guarantee accuracy by construction: model cascades and approximate query processing techniques bound accuracy relative to the unrewritten plan [24, 25, 38, 58]. However, all approaches require some notion of ground truth—either a user-specified "oracle" LLM-powered implementation [25, 38, 42], a user-defined accuracy function [42], or LLM-as-judge pairwise comparisons for ranking candidate plans [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 into optimal plans [18, 19]. For semantic operators, two new challenges arise: plans may lack optimal substructure (as explained in Sec. 1), and with accuracy as an objective, optimizers should return a Pareto frontier of plans, so users can choose their preferred cost-accuracy tradeoff. Only ABACUS, DocETL-V1, and MOAR search over a space of rewrites: ABACUS adapts Cascades [18] to return a Pareto frontier; DocETL-V1 introduces a top-down search algorithm designed for LLM-as-judge evaluation; MOAR uses UCT-based search [28] over complete pipelines, avoiding optimal substructure assumptions. ThalamusDB and LOTUS search for optimal parameters within specific rewrites (e.g., model cascade thresholds) but do not search across different rewrites. Other solutions provide interactive interfaces to help users rewrite pipelines themselves [34, 45].

Overall, MOAR spans the broadest rewrite space of any semantic query optimizer, returns a frontier of high-accuracy plans at low costs, and searches over rewrites of complete pipelines to handle the lack of optimal substructure in LLM-powered data processing.

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 [42], the next-best optimizer, MOAR achieves 27% higher accuracy on average while matching its best accuracy at only 55% of its cost.

Looking forward, two directions could improve the efficiency of MOAR's search process. First, reducing reliance on frontier (e.g., gpt-5) LLM agents for rewrite instantiation—e.g., through learned models or heuristics. Second, finding ways to estimate accuracy and cost without executing pipelines on samples. But overall, more broadly, MOAR demonstrates the promise of agentic approaches to query optimization: by delegating both the discovery and instantiation of rewrites to LLM agents guided by structured directives and intelligent search, we can navigate the vast space of possible query plans for LLM-powered data processing more effectively than traditional rule-based optimizers.

REFERENCES

[1] Asma Ben Abacha, Wen-wai Yim, Yujuan 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/cidrdb/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 döt 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] DataNeed. 2024. Company Reports Dataset: 5,436 ESG/Sustainability Reports. https://huggingface.co/datasets/DataNeed/company-reports. Accessed: 2025-09-19.

[14] 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

[15] Till Döhmen. 2024. Introducing the prompt() Function: Use the Power of LLMs with SQL! urlhttps://motherduck.com/blog/sql-llm-prompt-function-gpt-models/. Accessed: 2025-06-22.

[16] Karel D'Oosterlinck, François Remy, Johannes Deleu, Thomas Demeester, Chris Develder, Klim Zaporojets, Aneiss Ghodsi, Simon Ellershaw, Jack Collins, and Christopher Potts. 2023. BioDEX: Large-Scale Biomedical Adverse Drug Event Extraction for Real-World Pharmacovigilance. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 13425–13454.

[17] 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.

[18] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data(base) Engineering Bulletin* 18 (1995), 19–29. https://api.semanticscholar.org/CorpusID:260706023

[19] 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.

[20] 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].

[21] 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

[22] Dan Hendrycks, Collin Burns, Anya Chen, and Spencer Ball. 2021. CUAD: An Expert-Annotated NLP Dataset for Legal Contract Review. *NeurIPS* (2021).

[23] 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.

[24] 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.

[25] 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

[26] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing Neural Network Queries over Video at Scale. *Proceedings of the VLDB Endowment* 10, 11 (2017).

[27] Omar Khattab, Arnav 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*.

[28] Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*. Springer, 282–293.

[29] Alexander W. Lee, Justin Chan, Michael Fu, Nicolas Kim, Akshay Mehta, Deepti Raghavan, and Uğur Çetintemel. 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

[30] 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.

[31] 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

[32] Yiming Lin, Madelon Hulsebos, Ruiying Ma, Shreya Shankar, Sepanta Zeighami, Aditya G Parameswaran, and Eugene Wu. 2025. Querying Templatized Document Collections with Large Language Models. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2422–2435.

[33] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, et al. 2025. Palimpzest: Optimizing ai-powered analytics with declarative query processing. In *Proceedings of the Conference on Innovative Database Research (CIDR)*. 2.

[34] Chunwei Liu, Gerardo Vitagliano, Brandon Rose, Matthew Printz, David Andrew Samson, and Michael Cafarella. 2025. PalimpChat: Declarative and Interactive AI analytics. In *Companion of the 2025 International Conference on Management of Data*. 183–186.

[35] 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

[36] 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

[37] OpenAI. 2024. GPT-4o mini: advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/ Accessed: 2025-10-19.

[38] 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.

[39] 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).

[40] 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.

[41] Armin Ronacher. 2008. Jinja2 Templating Engine. https://jinja.palletsprojects.com/. Version 3.x. Accessed 2025-11-21.

[42] 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).

[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] Antoni Sobkowicz and Wojciech Stokowiec. 2016. Steam review dataset-new, large scale sentiment dataset. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016) Workshop Emotion and Sentiment Analysis*. 55–58.

[50] Zhaoze Sun, Chengliang Chai, Qiyan 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.

[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] Paul Viola and Michael Jones. 2001. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, Vol. 1. Ieee, I–I.

[54] 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

[55] Jiayi Wang and Guoliang Li. 2025. Aop: Automated and interactive llm pipeline orchestration for answering complex queries. CIDR.

[56] 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

[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] Barry Zhang, Keith Lazuka, and Mahesh Murag. 2025. Equipping agents for the real world with Agent Skills. https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills. Accessed: December 1, 2025.

[60] Eckart Zitzler. 1999. *Evolutionary algorithms for multiobjective optimization: Methods and applications*. Vol. 63. Shaker Ithaca.

APPENDIX

In this appendix, we provide the complete DocETL operator library (App. A), detailed descriptions of all new rewrite directives introduced in MOAR (App. B), pseudocode for the pipeline selection and rewriting algorithms (App. C), and additional experimental results including pipeline latencies, optimization overhead, and pairwise cost savings across all methods (App. D).

A OPERATORS IN DOCETL

DocETL provides a library of semantic and auxiliary (i.e., not parameterized by natural language) operators used to construct document processing pipelines. Table 7 summarizes all operators supported in our implementation.

B DETAILED DESCRIPTIONS OF NEW REWRITE DIRECTIVES

Throughout this section, we adopt the notation from Shankar et al. [44]: given operators A and B, we denote their composition as $A \to B$, where $(A \to B)(D) = B(A(D))$. For independent execution, we use $A \parallel B$. We may drop arguments (e.g., $\text{Map}_x(D)$ becomes Map_x) and omit subscripts except when the same operator appears multiple times. We color new or modified operators introduced by a rewrite in green. The arrow $\Rightarrow$ denotes a rewrite of the operator (or operator sequence) on the left into the form on the right.

New Directive Categories. Categories marked with † are entirely new to MOAR: operator fusion, approximation, reordering, and arbitrary rewrites.

Parameter-Sensitive Directives. Some directives are parameter-sensitive and marked with ‡: they generate multiple candidate rewrites with different parameter values, evaluate all candidates on the sample D_o, and select the one achieving highest accuracy. For these directives, we instruct the LLM agent to synthesize multiple distinct configurations exploring different trade-offs (e.g., precision vs. recall, cost vs. context length). Parameter-sensitive directives either enumerate discrete parameter values (e.g., chunk sizes) or use the agent to generate diverse configurations. The number of candidates generated is denoted by k in Algorithm 3.

B.1 Fusion and Reordering†

Operator fusion combines multiple sequential operators into fewer operators, reducing the number of LLM calls and avoiding redundant passes over the same document. Reordering re-arranges commuting operators so that cheaper or more selective operators run earlier, reducing the amount of work done by expensive operators. This category is new in MOAR and primarily targets cost reduction, subject to preserving pipeline semantics.

When instantiating these directives, the LLM agent synthesizes merged prompts that unify the semantics of fused operators, combined output schemas, and, when necessary, auxiliary logic to maintain compatibility with downstream operators. It also verifies that reordering preserves semantics (e.g., ensuring a filter does not depend on outputs from an operator it is moved before).

In general, any pair of adjacent operators of the same type (e.g., `Map–Map` or `Filter–Filter`) can be fused into a single operator. Special cases arise when the operators differ in type but have dependent semantics, such as `Map–Reduce`, `Map–Filter`, or `Filter–Map`.

B.1.1 Same-type Fusion This directive fuses adjacent operators of the same type (e.g., `Map–Map`, `Filter–Filter`, or `Reduce–Reduce`) into a single operator that implements the combined semantics:

$$\text{Map}_x \to \text{Map}_y \Rightarrow \text{Map}_z,$$

and similarly for filters and reduces. The LLM agent rewrites the prompt template in z to cover both tasks, and synthesizes an output schema that is the union of the original schemas (dropping intermediate fields that are not needed downstream). This reduces the number of LLM calls without requiring additional passes over the data.

B.1.2 Map–Reduce Fusion This directive fuses a map with a downstream reduce, eliminating the need to materialize intermediate results:

$$\text{Map}_x \to \text{Reduce}_{K,y} \Rightarrow \text{Reduce}_{K,z} \tag{2}$$

The LLM agent rewrites the prompt of $\text{Reduce}_{K,y}$ to also perform the logic described by the preceding Map_x. This allows the reduce to compute per-document transformations and aggregate their results within a single LLM call. For example, in Ex. 1.1, if police report documents already have a key or attribute representing case type, and the pipeline first maps each report to a list of enhancement factors (Map_x) and then summarizes them by case type ($\text{Reduce}_{K,y}$), the agent can rewrite the reduce prompt in z to both extract and summarize enhancement factors in one pass of the documents.

Note that a precondition for Eq. (2) to be invoked is that when the output schema of Map_x does not generate any of the grouping attributes in K; otherwise, the groupby keys would not exist prior to aggregation.

B.1.3 Map–Filter Fusion This directive fuses an LLM-powered map followed by an LLM-powered filter that depends only on the map's outputs:

$$\text{Map}_x \to \text{Filter}_y \Rightarrow \text{Map}_z \to \text{CodeFilter} \tag{3}$$

The Map_z operator has a rewritten prompt that incorporates the logic of both the original map and filter, and an extended output schema that incorporates the boolean attribute in y's output schema, indicating whether the document should be retained. The downstream CodeFilter operator is then programmatically synthesized to drop documents where this attribute is false, ensuring that the pipeline preserves the original filter semantics.

For example, in Ex. 1.1, if Map_x extracts snippets of text describing instances of excessive force and the Filter_y identifies those involving a firearm, the agent rewrites the map prompt in z to both extract the snippets *and* predict whether each involves a weapon, producing a boolean flag in addition to the output schema attributes requested in x. The code filter CodeFilter then removes entries where the flag is false.

B.1.4 Filter–Map Fusion This directive fuses an LLM-powered filter followed by an LLM-powered map, replacing the two-step evaluation with a single LLM call followed by a lightweight deterministic filter:

$$\text{Filter}_x \to \text{Map}_y \Rightarrow \text{Map}_z \to \text{CodeFilter} \tag{4}$$

Table 7: DocETL operator library. Semantic operators invoke an LLM; operators marked with * do not. Operators marked with † are new in MOAR.

Operator	User configuration	Description
`map`	prompt, output schema	Uses an LLLM to execute a per-document transformation, adding new keys to the schema (and optionally omitting existing ones).
`parallel-map`	multiple prompts, output schemas	Runs multiple independent `map` operations in parallel on each document, merging all resulting fields into the schema.
`reduce`	group-by keys, prompt, output schema	Uses an LLM to aggregate groups of documents sharing the same key values, producing one output document per group.
`filter`	boolean prompt	Uses an LLM to evaluate a boolean condition per document, retaining only documents for which the condition is true.
`resolve`	comparison prompt, resolution prompt	Uses an LLM to identify fuzzily matching values across documents and replace them with canonicalized versions through a two-step compare–resolve process.
`equijoin`	comparison prompt	Uses an LLM to semantically compare pairs of documents and determine whether they should be joined on fuzzy or contextual matching of key values.
`unnest*`	array/dict field	Flattens nested array or dictionary fields: arrays create multiple documents, while nested dicts are merged into parent documents.
`split*`	split key, chunk size	Divides documents into token-limited or rule-based chunks, producing one document per chunk.
`gather*`	context-window configuration	Augments each chunk with surrounding context (preceding and following chunks), without changing the number of documents.
`sample*†`	sampling method; sample size; optional query; stratification keys	Selects a subset of documents or chunks before downstream processing. The optional query is a text template (provided by the user or synthesized by the agent) that the sampler instantiates to assess relevance under BM25 or embedding-based sampling. Sampling can also be stratified on user-provided keys.
`extract†`	prompt returning line ranges	Uses an LLM to output only the relevant line spans ("lines 45–67, 103–120"), returning a "compressed" version or subset of the original document.
`code-map*†`	Python code, output schema	A code-powered version of `map`; runs a user- or agent-generated Python function over each document and produces outputs matching the specified schema.
`code-reduce*†`	Python code, output schema	A code-powered version of `reduce`; performs grouping and aggregation in Python, often followed by a light-weight `map` operator to generate narrative or structured summaries.
`code-filter*†`	Python code that returns true or false	A code-powered version of "filter": evaluates a Python boolean function on each document and discards those for which the function returns false.

The Map_z operator has a rewritten prompt that combines the logic of both the original filter and map, and extends the output schema with a boolean attribute indicating whether the document satisfies the filter condition in x. Like in Eq. (3), the downstream CodeFilter operator is programmatically synthesized. For example, in Ex. 1.1, if Filter_x identifies police reports describing violent incidents and Map_y extracts snippets of text describing the use of excessive force, the agent rewrites the map prompt to, at the same time, both extract these snippets and predict whether the incident qualifies as violent, outputting a boolean flag. CodeFilter then simply removes entries where the flag is false, avoiding a separate LLM call for filtering.

Note that Filter–Map fusion effectively "pulls up" the filter into the map stage. As a result, the rewrite may not always be optimal—especially when Filter_x can be executed with a cheaper model than Map_y, or when the selectivity of Filter_x is low enough that performing it separately would substantially reduce the number of documents processed by the more expensive map.

B.1.5 Reordering Inspired by classical query optimization techniques, this directive reorders commuting operators to improve efficiency by moving selective or shrinking operations earlier in the pipeline:

$$o_x \to o_y \Rightarrow o_y \to o_x.$$

MOAR applies this rewrite only when the LLM agent verifies that reordering preserves pipeline semantics, e.g., the rewritten filter does not depend on attributes produced by the operator it is moved before. In practice, user-authored pipelines are short (2–3 operators) [45], so reordering becomes more valuable as MOAR's search generates longer, more complex pipelines through sequences of rewrites.

B.2 Code Synthesis†

This new category directly targets cost reduction by replacing LLM calls with custom Python code intended to approximate the task described in the semantic operator.

B.2.1 Code Substitution This directive replaces an LLM-powered operator with synthesized Python code:

$$o_x \Rightarrow \text{Code}_{\hat{x}} \tag{5}$$

where Code is the code-powered version of the same operator type as o_x, and $\hat{x}$ contains Python code synthesized by an LLM agent to approximate the task specified in the original prompt template in x. The output schema remains unchanged—$\text{Code}_{\hat{x}}$ must produce outputs conforming to the same schema s as the original operator. For Ex. 1.1, if the task is to extract any mention of a firearm, the LLM agent might synthesize a regular expression that matches common firearm-related terms (e.g., gun, pistol, rifle, weapon, firearm, armed) and extracts surrounding sentences as context, avoiding LLM inference costs entirely while producing outputs in the same format as the original LLM-powered extraction.

B.2.2 Code Substitution (Reduce) A specialized version of code substitution targets reduce operations where parts of the task are better suited to code than LLMs:

$$\text{Reduce}_x \Rightarrow \text{CodeReduce}_{\hat{x}} \to \text{Map} \tag{6}$$

This directive splits the reduce task into two stages: deterministic aggregations handled by code, followed by text generation or reasoning handled by an LLM. Both operators are synthesized by the LLM agent. For Ex. 1.1, suppose the original reduce (Reduce_x) generates a detailed report summarizing enhancement factors per officer, including total counts and breakdowns by type. The agent might split this into: *(i)* a CodeReduce that groups documents by officer

name, counts total enhancement factors, and computes counts per type (firearm, injury, kidnapping), producing structured data; and *(ii)* a Map that generates narrative report text from this data (e.g., "Person X had 2 total enhancement factors: one firearm-related, one injury-related").

B.2.3 Document Compression (Code-based)‡ When the relevant content can be identified through deterministic rules rather than semantic understanding, this directive replaces the LLM-powered extraction with synthesized code, following the template:

$$o_x \Rightarrow \text{CodeMap} \rightarrow o_{x'} \tag{7}$$

The CodeMap operator executes Python code (using only standard libraries and the regular expression library `re`) that returns a compressed version of the document. This document compression approach avoids LLM calls entirely, unlike the previous approaches in Eqs. (11) and (12). The operator $o_{x'}$ is a modified version of o_x whose prompt references the compressed content. For Ex. 1.1, if enhancement factors are always mentioned in sections with specific headers (e.g., "Incident Details," "Evidence Collected"), the LLM agent might synthesize a code map that uses regular expressions to extract only paragraphs under these headers. The agent synthesizes both the code for CodeMap and the modified prompt for $o_{x'}$.

This directive is parameter-sensitive. We instruct the LLM agent to synthesize two entirely different code implementations exploring different trade-offs: one optimizing for precision (stricter pattern matching) and one optimizing for recall (broader pattern matching). For example, when extracting firearm mentions, the precision-focused implementation might match only explicit weapon terms with exact regular expressions, while the recall-focused implementation might include broader contextual patterns and proximity-based matching. Both implementations are evaluated on D_o and the higher-accuracy variant is selected.

B.2.4 Head/Tail Compression‡ One specific instantiation of code-based document compression extracts only the first h words (head) and last ℓ words (tail) from a document, which we refer to as *head-/tail* compression. While head/tail compression follows the same template in Eq. (7), we provide it as an explicit directive because agents do not reliably discover it independently, even though the pattern is broadly applicable. For example, classifying a document's genre or identifying its author may only require examining the opening paragraphs.

This directive is parameter-sensitive. In the directive description provided to the agent (as explained in Sec. 3), we instruct the agent to generate two different configurations with different head/tail lengths: one using shorter context windows (e.g., $h = 100, \ell = 50$) optimizing for cost efficiency, and another using longer windows (e.g., $h = 300, \ell = 150$) optimizing for higher recall. For tasks where key information appears in opening paragraphs (e.g., document classification or author identification), the directive description (as explained in Sec. 3) suggests the agent may allocate more words to the head. Both configurations are evaluated on D_o and the higher-accuracy variant is selected.

As LLM agents improve, or we train our own agents based on known rewrite directive patterns, explicit instantiations of directives like head/tail compression may become unnecessary. But, for now, they provide valuable guidance for discovering common optimization patterns.

B.3 Data Decomposition

DocETL introduced directives for document chunking, to improve accuracy when processing long documents, and for multi-level aggregation, to combine results across groups of documents. MOAR extends the "data decomposition" category with additional chunking strategies that provide more fine-grained control over *which* portions of documents to process.

B.3.1 Chunk Sampling‡ When documents are split into many chunks, processing all chunks may be unnecessary if only a subset contains relevant information. The chunk sampling directive introduces a sampling step after gathering context:

$$\text{Split} \rightarrow \text{Gather} \rightarrow \text{Map} \rightarrow \text{Reduce} \Rightarrow \text{Split} \rightarrow \text{Gather} \rightarrow \text{Sample} \rightarrow \text{Map} \rightarrow \text{Reduce} \tag{8}$$

The Sample operator can be instantiated with random sampling, keyword search (based on BM25 retrieval [40]), or embedding-based similarity, selecting the top-k document chunks relevant to a query [30]. When instantiating the directive, the LLM agent synthesizes the sampling method, the query (if not random sampling), and the value of k.

For example, in Ex. 1.1, the agent might choose keyword-based sampling after splitting police records into chunks. It could synthesize the keyword list `["firearm", "injury", "kidnapping", "weapon", "harm"]` and set $k = 20$ to select the 20 chunks with highest BM25 scores before applying the extraction map. Alternatively, for identifying cases involving excessive force, the agent might choose embedding-based sampling with the query "excessive use of force by police officer," computing embeddings for all chunks and selecting the k chunks with highest cosine similarity to this query. These selected chunks are then processed by the map and their results reduced into a final output.

This directive is parameter-sensitive. We instruct the LLM agent to generate two entirely different sampling configurations: one optimizing for precision (using stricter sampling criteria with lower k) and one optimizing for recall (using broader sampling criteria with higher k). For example, the precision-focused configuration might use BM25 with $k = 10$ and strict keyword matching, while the recall-focused configuration might use embedding-based sampling with $k = 30$ and a broader text query. Both configurations are evaluated on D_o and the higher-accuracy variant is selected.

B.3.2 Document Sampling‡ When a reduce operation aggregates over many documents within each group (where the grouping is defined by the reduce keys), it may be unnecessary to process every document if many contribute little or no signal to the final aggregation. The document sampling directive inserts a sampling stage before the reduce:

$$\text{Reduce}_{K,x} \Rightarrow \text{Sample}_K \rightarrow \text{Reduce}_{K,x}. \tag{9}$$

The operator Sample_K selects a subset of documents from each group defined by K, using random sampling, BM25 keyword search, or embedding-based similarity. The LLM agent synthesizes both

the sampling method and the per-group sample size k, selecting the documents most relevant to the downstream aggregation logic.

For example, in Ex. 1.1, suppose the pipeline aggregates enhancement factors per precinct, where $K = \{\texttt{precinct\_id}\}$. Some precincts may contain hundreds of reports, many of which have no enhancement-related content. The agent might synthesize an embedding-based sampler that, for each precinct, selects the $k = 30$ reports most similar to the query "mentions of injuries, weapons, or threats," forwarding only these to the reduce. A precision-oriented configuration might instead select the $k = 10$ reports containing explicit weapon or injury keywords ("firearm", "injury", "harm", "weapon") using BM25.

This directive is parameter-sensitive. We instruct the agent to generate at least two distinct sampling configurations—one emphasizing precision (smaller k, stricter criteria) and one emphasizing recall (larger k, broader retrieval). Both variants are evaluated on D_o, and the higher-accuracy configuration is selected.

B.3.3 Cascade Filtering‡ This directive optimizes filtering costs by injecting a cascade of cheaper "pre-filters" before an expensive LLM-powered filter:

$$\text{Filter}_x \Rightarrow \text{CodeFilter} \rightarrow \text{Filter}_y \rightarrow \text{Filter}_x \tag{10}$$

where one or more code filters and LLM filters may be synthesized. The cascade consists of two stages of pre-filters, ordered by increasing cost: first, deterministic Python code (using regular expressions, keyword matching, or simple logic) that quickly eliminates documents failing obvious criteria; second, cheap LLM-powered filters Filter_y with simplified prompts and inexpensive models (e.g., gpt-5-nano), ordered by prompt length (shortest first). The pre-filters prioritize high recall (rarely rejecting documents that would pass the main filter) but may have lower precision (allowing through documents that will eventually be filtered out). This design ensures the final filter produces the same results as the original, while reducing cost by eliminating many documents before expensive evaluation.

When instantiating this directive, the LLM agent examines sample documents from D_o to identify patterns distinguishing documents that pass versus fail the main filter. The agent then synthesizes code filters for patterns observable through keyword presence, regular expressions, or document structure, and LLM filters with short prompts that evaluate simple semantic properties difficult to capture with code. For Ex. 1.1, if the original filter identifies police reports describing violent incidents with firearms, the agent might synthesize: (i) a code filter checking for weapon-related keywords ("gun", "pistol", "firearm", "weapon"); (ii) a gpt-5-nano filter checking "Does this report describe a violent incident?"; followed by (iii) the original expensive filter performing nuanced interpretation of what constitutes a violent firearm incident.

This directive is parameter-sensitive: we instruct the agent to generate two cascade configurations exploring different combinations of code filters and LLM pre-filters, evaluating each on D_o to select the highest-accuracy pipeline.

B.4 Projection Synthesis

DocETL introduced projection synthesis directives that decompose complex tasks into simpler subtasks (e.g., chaining multiple maps, isolating independent projections). MOAR extends this category by identifying a sub-class of projection synthesis: rather than decomposing the *task* described in an operation's prompt, these directives reduce the *data* processed by the operation. By making documents smaller before applying an LLM-powered operator, these directives can improve cost while preserving the information needed for accurate results. We provide various directives to make the documents smaller.

B.4.1 Document Summarization This directive inserts a map operation at the beginning of the pipeline to summarize each document, preserving all information needed for downstream operations, following the template:

$$o_x \Rightarrow \text{Map} \rightarrow o_{x'} \tag{11}$$

Here, Map produces a summary of the document, and $o_{x'}$ is a modified version of the original operator o_x whose prompt references the summary instead of the full document. The LLM agent synthesizes both the summarization operator Map and the modified prompt for $o_{x'}$ to ensure all information needed by the downstream extraction is preserved.

B.4.2 Document Compression (LLM-based) This directive inserts an LLM-powered *extraction* operation at the beginning of the pipeline to retain only content relevant for downstream operations, following the template:

$$o_x \Rightarrow \text{Extract} \rightarrow o_{x'} \tag{12}$$

The Extract operator asks the LLM to output line ranges that are relevant (e.g., "lines 45-67, 103-120"), which are then converted back into a subset of the original document. This differs from summarization in Eq. (11), where the map operation generates entirely new text—which might not be a subset of the original document. Since Extract outputs only line ranges rather than narrative text, it is typically cheaper to execute. The operator $o_{x'}$ is a modified version of o_x whose prompt references the extracted content. The LLM agent synthesizes both the extraction prompt and the modified prompt for $o_{x'}$.

B.5 LLM-Centric Rewrites

DocETL introduced directives that improve LLM output quality by refining how tasks are specified to the LLM. For example, the gleaning directive uses a validator LLM to check outputs and provide feedback for iterative refinement. Similar strategies have been explored in ABACUS, which implements a "critique-and-refine" physical implementation of map operations [42]. Inspired by prompting strategies [35] and optimization techniques [39], MOAR adds directives that improve prompt quality and provide examples to guide LLM behavior.

B.5.1 Model Substitution This directive replaces the model used by an operator:

$$o_x \Rightarrow o_{x'} \tag{13}$$

where $x = (t, s, m)$ and $x' = (t, s, m')$ with $m' \neq m$. When instantiating this directive, the LLM agent receives context about model performance: for each model in the available pool M, the agent sees the cost and accuracy achieved by the original pipeline when executed with that model on a sample of data. The agent also has access to each model's performance on MRCR (a long-context benchmark

that evaluates an LLM's ability to retrieve and distinguish between multiple similar requests hidden in long contexts [36, 54]), as well as context window size and pricing details.

Using this information, the agent can reason about model capabilities and select m' based on the operator's complexity and position in the pipeline. For Ex. 1.1, the agent might substitute GPT-4o-mini for operators extracting explicit mentions of weapons, while keeping GPT-4o for operators requiring more complex interpretation (e.g., determining if force was excessive given the circumstances).

B.5.2 Clarify Instructions in Prompt‡ This directive rewrites an operator by making its prompt template more specific and detailed:

$$o_x \Rightarrow o_{x'} \tag{14}$$

where $x = (t, s, m)$ and $x' = (t', s, m)$, with t' being a clarified version of prompt template t. The LLM agent analyzes the original prompt and a sample of documents, identifies ambiguous instructions, and generates a more detailed prompt that reduces the likelihood of misinterpretation. For Ex. 1.1, an original prompt might say *"extract evidence of threatening with a firearm."* After examining sample police reports, the agent might observe that reports use varied terminology ("weapon," "gun," "pistol," "armed") and describe threats in different ways ("pointed at," "brandished," "displayed"). The agent might then clarify the prompt to *"extract evidence of threatening with a firearm. This includes any instance where:* (i) *the report mentions a firearm, weapon, gun, pistol, rifle, or other projectile weapon; AND* (ii) *the report describes the weapon being pointed at, brandished, displayed, or used to intimidate someone. Extract the complete sentence(s) containing both elements."*

This directive is parameter-sensitive. We instruct the LLM agent to generate two different clarified prompts exploring different clarification strategies, and both clarified prompts are evaluated on D_o to determine which clarification strategy yields higher accuracy.

One might wonder why such clarifications help—why can't the LLM executing the operator simply reason through these ambiguities? In practice, the LLM agent performing optimization (e.g., gpt-5) is typically more powerful than the models used in operations (i.e., m, which might be gpt-4o-mini for cost efficiency). The more capable agent is therefore well-suited to identify ambiguities and aspects requiring additional reasoning, then encode that reasoning directly into the prompt so that cheaper models can execute the task reliably.

B.5.3 Find Few-shot Examples A popular prompt engineering technique is to include few-shot examples in the prompt to demonstrate desired behavior [8]. This directive synthesizes such examples to improve operator accuracy:

$$o_x \Rightarrow o_{x'} \tag{15}$$

where $x' = (t', s, m)$ and t' is a modified prompt template that incorporates few-shot examples. The agent examines sample documents, generates input-output pairs that demonstrate the desired behavior, and constructs the modified prompt with these examples embedded. Note that tools like DSPy [27] could automate few-shot example generation when this directive is chosen. However, DSPy's iterative optimization process (which evaluates multiple example sets to find the best one) takes significantly longer than our agent's "single-pass" instantiation. Supporting such iterative optimization would require modifying our search algorithm to account for varying directive instantiation costs—we leave this to future work.

B.5.4 Arbitrary Rewrite Beyond the structured directives described above, MOAR supports a "meta-directive" that allows LLM agents to propose custom rewrites without any directive scaffolding. This flexibility is important because the space of possible rewrites is unbounded, and even a large directive library cannot anticipate all beneficial transformations. For Ex. 1.1, an agent might propose adding a new map operation that extracts the reporting officer's precinct and experience level from metadata fields, then uses this information in downstream extraction prompts to adjust interpretation of what constitutes "excessive force" based on departmental policies and officer training. This would require edits to multiple different operators spread across the pipeline, and does not fit cleanly into any existing directive category.

To implement arbitrary rewrites, we pass the entire pipeline as YAML code to the agent and ask it to propose edits using a find-and-replace approach inspired by how coding agents work [3]. Specifically, the agent returns a list of search-and-replace blocks, where each block specifies: *(i)* a unique string to search for in the original pipeline YAML, and *(ii)* the replacement text. After the agent produces an arbitrary rewrite, we verify that the resulting pipeline can be parsed and executed by DocETL. If parsing or validation fails, we provide the error message to the agent and retry up to 3 times before discarding the rewrite.

C SEARCH ALGORITHM DETAILS

This appendix provides pseudocode for two of MOAR's search procedures. Algorithm 2 describes the selection phase, which traverses the search tree using UCT-based utility scores with progressive widening to choose which pipeline to rewrite next. Algorithm 3 describes the rewriting and evaluation phase, which uses an LLM agent to choose and instantiate a directive, then evaluates the resulting pipeline(s) on the sample D_o.

Algorithm 2: Selecting the pipeline to rewrite

Input: Root pipeline P_0, current search tree T_t
Output: Pipeline $P^\star$ to rewrite

Function `Select`(P_0, G_t)**:**
  $P \leftarrow P_0$;
  while *true* **do**
    if $|\text{children}(P)| < W(n_t(P))$ ***or*** *P has no evaluated children* **then**
      break;
    `// Descend to child with highest utility`
    $P \leftarrow$ child $P' \in \text{children}(P)$ with highest $U_t(P')$;
  end
  `// Increment visit count for` P `and all ancestors`
  $P' \leftarrow P$;
  while $P' \neq$ null **do**
    $n_t(P') \leftarrow n_t(P') + 1$;
    $P' \leftarrow \text{parent}(P')$;
  end
  return P;

Algorithm 3: Rewriting and evaluation

Input: Selected pipeline $P^\star$, current search tree $T_t = (V_t, E_t)$, directive usage map ν, evaluation sample D_o
Output: Evaluated child pipeline P', applied rewrite r, statistics $(\hat{c}(P'), \hat{a}(P'))$, and candidate count k

Function `RewriteAndEvaluate`$(P^\star, G_t, \nu, D_o)$**:**
  `// Determine objective based on frontier position`
  rank $\leftarrow$ rank of $P^\star$ by accuracy among pipelines in V_t;
  if *rank* $\leq |V_t|/2$ **then**
    objective $\leftarrow$ "reduce cost while preserving accuracy";
  else
    objective $\leftarrow$ "improve accuracy";
  end
  `// Step 1: Prune registry and choose directive`
  allowed_directives $\leftarrow$ PruneRegistry($P^\star$, registry, E_t);
  $(d, \text{target\_ops}) \leftarrow$ ChooseDirective($P^\star$, allowed_directives, T_t, ν, $\{\mu_t(\cdot)\}$, objective);
  $\nu(P^\star, d) \leftarrow \nu(P^\star, d) + 1$; `// Soft-prevent other concurrent workers choosing this directive`
  `// Step 2: Instantiate directive (may generate multiple candidates for parameter-sensitive directives)`
  $\{r_1, \ldots, r_k\} \leftarrow$ InstantiateDirective(d, target_ops, $P^\star$, objective, D_o);
  `// Evaluate all candidates and select most accurate`
  for $i = 1$ *to* k **do**
    $P'_i \leftarrow$ apply rewrite r_i to $P^\star$;
    execute P'_i on D_o to obtain $(\hat{c}(P'_i), \hat{a}(P'_i))$;
  end
  $(r, P') \leftarrow \arg\max_{(r_i, P'_i)} \hat{a}(P'_i)$; `// Select rewrite with highest accuracy`
  return $(P', r, \hat{c}(P'), \hat{a}(P'), k)$;

D ADDITIONAL EXPERIMENTAL RESULTS

This appendix provides supplementary experimental data. Table 8 reports test-time latencies for each method's optimized pipelines. Table 9 compares optimization costs and latencies across all methods. Fig. 6 presents pairwise cost savings matrices showing how much each method costs to match every other method's accuracy, both for best-accuracy pipelines and averaged across all Pareto-optimal pipelines.

Dataset	MOAR	Original	SA	LOTUS	PZ	PZ-d	PZ-r&r	LOTUS-d	LOTUS-r&r	DocETL-V1
CUAD	179.24 ± 134.64	89.91	160.78 ± 126.64	92.18	60.65 ± 9.82	–	–	–	–	140.00
Game Reviews	88.04 ± 42.85	350.98	279.19 ± 8.05	1446.50	735.57 ± 414.51	–	–	–	–	240.85
BlackVault	92.68 ± 28.51	34.89	41.70 ± 1.51	23.34	–	–	–	–	–	37.14
Biodex	163.31 ± 123.13	150.25	245.00 ± 274.87	–	–	205.49 ± 253.37	464.34 ± 20.20	307.17	12.15	402.22
Medec	51.76 ± 48.67	6.95	50.10 ± 78.30	28.90	39.90 ± 11.51	–	–	–	–	6.52
Sustainability	254.72 ± 218.29	109.91	78.57 ± 7.57	2094.90	–	–	–	–	–	686.52

Table 8: Test plan latency (seconds) across datasets and methods. "Original" refers to the user-specified pipeline prior to optimization. For methods returning multiple pipelines, values show mean ± std across all discovered pipelines; single values indicate one pipeline. "–" indicates the method is not evaluated on that dataset.

	Optimization Cost						Optimization Latency (s)				
Dataset	MOAR	SA	PZ	PZ-d	PZ-r&r	DocETL-V1	MOAR	SA	PZ	PZ-d	PZ-r&r
CUAD	$44.04	$0.34	$54.87	–	–	$1.58	5535.6	156.62	11579.34	–	–
Game Reviews	$57.29	$0.29	$28.82	–	–	$6.60	8382.16	2501.66	19718.02	–	–
BlackVault	$35.44	$0.24	–	–	–	$1.84	2353.17	1091.66	–	–	–
Biodex	$188.2	$8.94	–	$93.24	$6.26	$14.35	3908.04	756.40	–	9711.17	1767.83
Medec	$16.73	$0.27	$4.59	–	–	$0.01	2889.4	166.4	3066.15	–	–
Sustainability	$79.21	$2.59	–	–	–	$ 43.7	2070.04	547.06	–	–	–

Table 9: Optimization Cost and Latency Comparison. "–" indicates the method is not evaluated on that dataset. Optimization cost and latencies are not reported for LOTUS, and optimization latencies are not reported for DocETL-V1.

Best Cost Savings Matrix

CUAD

Base \ Target	MOAR	Original	SA	LOTUS	PZ	DocETL-V1
MOAR	–	0.912×	0.377×	–	0.290×	0.073×
Original	n/a	–	n/a	–	n/a	n/a
SA	n/a	0.925×	–	–	n/a	0.194×
LOTUS	n/a	1.099×	n/a	–	n/a	n/a
PZ	n/a	3.136×	1.296×	–	–	0.252×
DocETL-V1	n/a	12.466×	n/a	–	n/a	–

Game Reviews

Base \ Target	MOAR	Original	SA	LOTUS	PZ	DocETL-V1
MOAR	–	0.067×	–	0.071×	0.003×	0.072×
Original	n/a	–	n/a	n/a	n/a	1.089×
SA	n/a	0.984×	–	–	n/a	n/a
LOTUS	n/a	0.938×	–	–	n/a	1.021×
PZ	n/a	8.119×	–	8.659×	–	8.838×
DocETL-V1	n/a	0.919×	–	0.980×	n/a	–

BlackVault

Base \ Target	MOAR	Original	SA	LOTUS	DocETL-V1
MOAR	–	0.146×	0.497×	–	0.267×
Original	n/a	–	n/a	–	n/a
SA	n/a	0.772×	–	–	1.407×
LOTUS	n/a	n/a	n/a	–	n/a
DocETL-V1	n/a	0.548×	n/a	–	–

Biodex

Base \ Target	MOAR	Original	SA	LOTUS-d	LOTUS-r&r	PZ-d	PZ-r&r	DocETL-V1
MOAR	–	0.424×	0.196×	0.145×	–	–	1.840×	0.152×
Original	n/a	–	n/a	n/a	–	–	n/a	n/a
SA	n/a	0.090×	–	0.337×	–	–	45.694×	0.353×
LOTUS-d	n/a	2.920×	n/a	–	–	–	n/a	1.047×
LOTUS-r&r	n/a	3.267×	n/a	n/a	–	–	n/a	n/a
PZ-d	n/a	n/a	n/a	n/a	–	–	n/a	n/a
PZ-r&r	n/a	0.232×	n/a	0.080×	–	–	–	0.083×
DocETL-V1	n/a	2.788×	n/a	0.955×	–	–	n/a	–

Medec

Base \ Target	MOAR	Original	SA	LOTUS	PZ	DocETL-V1
MOAR	–	1.029×	0.966×	1.245×	0.046×	0.840×
Original	n/a	–	n/a	1.210×	0.044×	0.816×
SA	1.035×	0.789×	–	0.954×	0.035×	0.644×
LOTUS	n/a	0.827×	n/a	–	0.037×	0.675×
PZ	n/a	22.594×	n/a	27.327×	–	18.439×
DocETL-V1	n/a	1.225×	n/a	1.482×	0.054×	–

Sustainability

Base \ Target	MOAR	Original	SA	LOTUS	DocETL-V1
MOAR	–	0.016×	0.143×	–	0.133×
Original	n/a	–	8.940×	–	n/a
SA	n/a	0.078×	–	–	n/a
LOTUS	n/a	3.985×	n/a	–	n/a
DocETL-V1	7.502×	3.436×	30.721×	–	–

Average Cost Savings Matrix

CUAD

Base \ Target	MOAR	Original	SA	LOTUS	PZ	DocETL-V1
MOAR	–	0.912×	0.630×	–	0.095×	0.073×
Original	n/a	–	1.04×	–	n/a	n/a
SA	n/a	0.925×	–	–	n/a	0.194×
LOTUS	n/a	1.099×	n/a	–	n/a	n/a
PZ	1.474×	3.136×	2.166×	–	–	0.252×
DocETL-V1	n/a	12.466×	12.968×	–	n/a	–

Game Reviews

Base \ Target	MOAR	Original	SA	LOTUS	PZ	DocETL-V1
MOAR	–	0.067×	–	0.071×	0.004×	0.072×
Original	n/a	–	n/a	n/a	n/a	1.089×
SA	n/a	0.984×	–	–	n/a	n/a
LOTUS	n/a	0.938×	–	–	0.115×	1.021×
PZ	n/a	8.119×	–	8.659×	–	8.838×
DocETL-V1	n/a	0.919×	–	0.980×	n/a	–

BlackVault

Base \ Target	MOAR	Original	SA	LOTUS	DocETL-V1
MOAR	–	0.146×	0.497×	–	0.267×
Original	n/a	–	n/a	–	n/a
SA	5.271×	0.772×	–	–	1.407×
LOTUS	n/a	n/a	n/a	–	n/a
DocETL-V1	n/a	0.548×	n/a	–	–

Biodex

Base \ Target	MOAR	Original	SA	LOTUS-d	LOTUS-r&r	PZ-d	PZ-r&r	DocETL-V1
MOAR	–	0.424×	0.216×	0.145×	–	–	0.515×	0.152×
Original	n/a	–	1.016×	n/a	–	–	n/a	n/a
SA	6.901×	0.090×	–	0.337×	–	–	10.481×	0.353×
LOTUS-d	6.881×	2.920×	2.967×	–	–	–	1.631×	1.047×
LOTUS-r&r	n/a	3.267×	n/a	n/a	–	–	n/a	n/a
PZ-d	n/a	n/a	n/a	n/a	–	–	n/a	n/a
PZ-r&r	0.546×	0.232×	0.236×	0.080×	–	–	–	0.083×
DocETL-V1	6.571×	2.788×	2.833×	0.955×	–	–	1.558×	–

Medec

Base \ Target	MOAR	Original	SA	LOTUS	PZ	DocETL-V1
MOAR	–	1.029×	1.086×	1.245×	0.046×	0.840×
Original	n/a	–	0.993×	1.210×	0.044×	0.816×
SA	2.145×	0.789×	–	0.954×	0.035×	0.644×
LOTUS	n/a	0.827×	0.821×	–	0.037×	0.675×
PZ	n/a	22.594×	51.051×	27.327×	–	18.439×
DocETL-V1	n/a	1.225×	1.216×	1.482×	0.054×	–

Sustainability

Base \ Target	MOAR	Original	SA	LOTUS	DocETL-V1
MOAR	–	0.016×	0.143×	–	0.133×
Original	n/a	–	8.940×	–	n/a
SA	n/a	0.078×	–	–	n/a
LOTUS	n/a	3.985×	n/a	–	n/a
DocETL-V1	11.351×	3.436×	30.721×	–	–

Figure 6: Comparison of cost savings across *all pairs* of methods on all workloads. Original represents the initial user-authored pipeline. Rows indicate the base method and columns indicate the target method. In the Best Cost Savings Matrix, each cell shows the monetary cost (in multiples) incurred by the base method to achieve the target method's best accuracy. In the Average Cost Savings Matrix, each cell shows the average monetary cost incurred by the base method to achieve the accuracy of each of the target method's pipelines. "n/a" indicates the base method cannot achieve the target method's accuracy; "–" indicates the target method does not achieve the original pipeline's accuracy. Diagonal entries are marked with "–".