DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing

Shreya Shankar[1], Tristan Chambers[2], Tarak Shah[2], Aditya G. Parameswaran[1], Eugene Wu[3]
[1]UC Berkeley EECS, [2]BIDS Police Records Access Project, [3]Columbia University
{shreyashankar,tristan.chambers,tarak_shah,adityagp} @berkeley.edu, ewu@cs.columbia.edu

ABSTRACT

Analyzing unstructured data has been a persistent challenge in data processing. Recent proposals offer declarative frameworks for LLM-powered processing of unstructured data, but they typically execute user-specified operations as-is in a single LLM call—focusing on cost rather than accuracy. This is problematic for complex tasks, where even well-prompted LLMs can miss relevant information. For instance, reliably extracting *all* instances of a specific clause from legal documents often requires decomposing the task, the data, or both.

We present DocETL, a system that optimizes complex document processing pipelines, while accounting for LLM shortcomings. DocETL offers a declarative interface for users to define such pipelines and uses an agent-based approach to automatically optimize them, leveraging novel agent-based rewrites (that we call *rewrite directives*), as well as an optimization and evaluation framework. We introduce *(i)* logical rewriting of pipelines, tailored for LLM-based tasks, *(ii)* an agent-guided plan evaluation mechanism, and *(iii)* an optimization algorithm that efficiently finds promising plans, considering the latencies of LLM execution. Across four real-world document processing tasks, DocETL improves accuracy by 21–80% over strong baselines. DocETL is open-source at `docetl.org` and, as of March 2025, has over 1.7k GitHub stars across diverse domains.

PVLDB Reference Format:
Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, Eugene Wu. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. PVLDB, 18(9): 3035 - 3048, 2025.
doi:10.14778/3746405.3746426

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ucbepic/docetl.

1 INTRODUCTION

Large Language Models (LLMs) have taken the world of data management by storm, with applications ranging from data integration, to tuning, to query optimization, to data cleaning [12]. There has also been an interest, all in the last few months, in declarative approaches to process unstructured data using LLMs [1, 29, 30, 38]. These systems, instrumented as extensions to the relational model for processing textual columns, typically assume the text snippets per row are *small and easy to process*. They therefore focus on reducing cost, while keeping accuracy almost the same. However, for many real-world tasks, that we refer to as ***complex document processing*** tasks, accuracy can be a significant bottleneck, limiting practical utility. Here, complexity can stem from the documents or the nature of the processing task, or both. Consider this scenario from our collaborators on the Police Records Access Project[1]:

Example 1.1 (Police Misconduct Identification). Journalists at Berkeley's Investigative Reporting Program want to analyze a large corpus of heterogeneous police records, obtained through records requests, to uncover patterns of misconduct and procedural violations. Records include police reports, court transcripts, internal affairs and medical examiner reports, and other case files, often spanning hundreds of pages each. Analysis involves extracting key information from long documents, aggregating information across documents to identify behavioral patterns for each officer, and generating summaries highlighting concerning trends.

Example 1.1 is representative of complex document processing tasks across domains including law, medicine, and social science. Consider a simpler version of this task, where we just want a summary of the role of each officer mentioned in each complex police record document, each with hundreds of pages. This task can be expressed as a single-step *map* operation applied to the OCR output per document, in one LLM call, with a user-provided prompt defining terms like "misconduct." All existing systems [1, 29, 30, 38] would simply execute the map operation, as is, with one LLM call per document. That is, they ***assume user-defined operations will yield sufficiently accurate results when executed by the LLM***, and focus primarily on reducing cost. However, this map operation may provide poor accuracy for multiple reasons. First, the document in question may exceed the LLM's context limit. Even if it fits, outputs may omit certain instances of misconduct, or include spurious information. Recent work has shown that LLM performance degrades considerably as length increases [27], because they can be distracted [47] or selectively pay attention to certain portions [31], failing to gain a holistic understanding [4, 22, 49, 56]. Simultaneous theoretical work has shown that this degradation is due to limits in the transformer architecture [23, 39, 48]. While one could apply prompt compilation [26, 54] to identify a better prompt, this relies on examples, which are either not present or are too long to include (e.g., an example document with hundreds of pages)—but irrespective do not fix the underlying challenges with LLMs performing a complex task on complex documents.

Our key insight is that the quality of LLM outputs is often not adequate for complex data processing—we cannot simply treat the

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746426

[1]https://bids.berkeley.edu/california-police-records-access-project

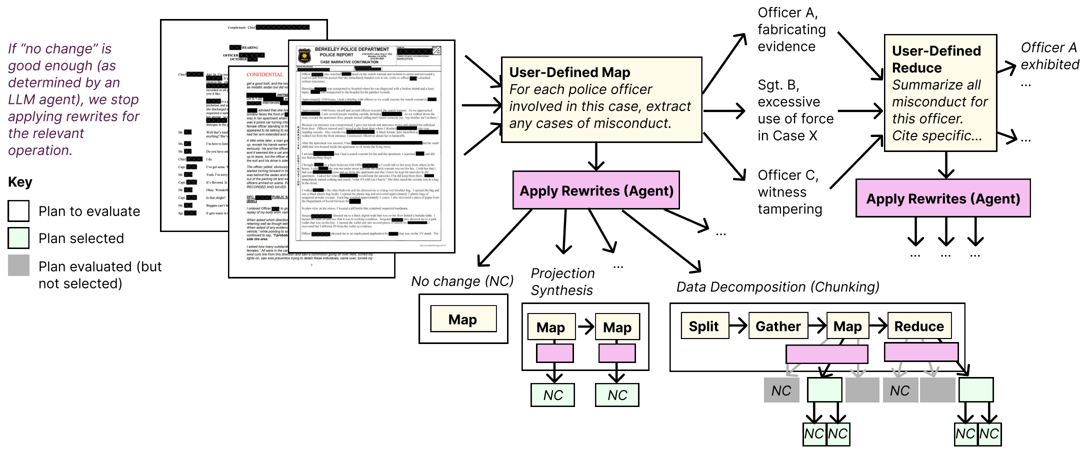

If "no change" is good enough (as determined by an LLM agent), we stop applying rewrites for the relevant operation.
Key
Plan to evaluate
Plan selected
Plan evaluated (but not selected)
BERKELEY POLICE DEPARTMENT
POLICE REPORT
CONFIDENTIAL
User-Defined Map
For each police officer involved in this case, extract any cases of misconduct.
Apply Rewrites (Agent)
No change (NC)
Map
Projection Synthesis
Map
Map
NC
NC
Data Decomposition (Chunking)
Split
Gather
Map
Reduce
NC
NC
NC
NC
NC
NC
Officer A, fabricating evidence
Sgt. B, excessive use of force in Case X
Officer C, witness tampering
User-Defined Reduce
Summarize all misconduct for this officer. Cite specific...
Officer A exhibited ...
Apply Rewrites (Agent)

Figure 1: Optimization for a pipeline designed to accomplish the task in Example 1.1. The diagram illustrates the system mid-optimization of the initial map operation. DocETL employs LLMs to synthesize new plans using novel rewrite directives. The process begins with an LLM verifier determining if an operation is sufficiently optimized. If not, rewriting continues. Notably, when a new operation is synthesized as part of a rewrite, it undergoes immediate opportunistic optimization, as shown by the nested "Apply Rewrites (Agent)" rectangles.

existing user-provided operators as fixed. Instead, ***we need to consider novel rewrites that decompose complex but error-prone operation(s) into a sequence of simpler and more accurate operations.*** For our map example, a different sequence of operations may increase accuracy. One such example is *map* ⟶ *map*, where the first map is tasked with removing all portions of each input document that do not pertain to misconduct (e.g., medical reports), while the second map is the single-step map above. Or we could replace the first map with one that summarizes each sequence of k paragraphs into one, keeping the second map as is. Yet another option is to replace the single-step *map* with what we call *split* ⟶ *gather* ⟶ *map* ⟶ *reduce*—a pattern that first *splits* the document into contiguous chunks; then, for each chunk, *gathers* k neighboring chunks before/after as context or background to be included into a prompt, generates per-officer summaries using its $2k$ neighbors as background context (*map*); and finally, performs a global summarization across all chunks (*reduce*).

However, ***we cannot expect a user to rewrite their pipeline into multiple alternatives and determine the one with the best performance***. The previous paragraph introduced three out of a multitude of potential rewrites, each of which could be recursively applied to operators in a pipeline, presenting a seemingly infinite set of options. For example, for the *map* ⟶ *map* pipeline, there are many alternatives for what the first map could do, and many different associated prompts. Even if we decide to use the first map to summarize k chunks at a time, determining the right value for k is challenging. Likewise for *split* ⟶ *gather* ⟶ *map* ⟶ *reduce*. Moreover, we're just focusing on the first step of the overall goal in Example 1.1, which is to summarize misconduct across all documents. So, we may need to apply a *reduce* operation across documents to group and summarize misconduct extractions by officer. However, the same officer may be extracted as "Officer Smith" in one document and "J. Smith" in another, resulting in separate, incomplete summaries for what should be a single officer [37]. It's not entirely clear how one would implement this form of entity resolution, and no current systems support it. In fact, additional context from the original document(s) may be necessary to determine if the two officers with the same name are identical. Finally, LLMs might struggle to recognize that multiple documents are from the same case, leading to overrepresentation of incidents in the misconduct summaries [52]. ***Overall, even an LLM expert would need extensive experimentation to design an accurate pipeline, given the dependency on the data, task, and LLM capabilities.*** This complexity underscores the need for a system that can automatically explore and evaluate different task decomposition strategies to find the most effective pipeline for a given task and dataset.

We present DocETL, our first attempt at developing a ***declarative system optimized for accurate complex document processing***. DocETL provides a declarative YAML-based interface for users to author pipelines with LLM-specific operators, including two new ones: *resolve* for entity resolution, and *gather* to maintain context when processing document chunks. Users can specify their pipeline at a high level with DocETL decomposing, rewriting, and optimizing the pipeline. DocETL introduces an *agent-based framework* to *rewrite* user-specified pipelines into alternative ones, as shown in Figure 1. Rather than simply relying on agents as-is, which can be error-prone, we *guide* them to rewrite query plans using novel ***rewrite directives*** that we identify. We call these directives instead of rules because they are abstract guidelines interpreted by LLMs based on task and data characteristics, with infinitely many concrete instantiations. We further leverage an agentic framework to *evaluate* the resulting pipelines. Since evaluation can be expensive, we develop an optimization approach inspired by Cascades [16], where we use a top-down rule-based strategy to generate and evaluate a space of equivalent plans, opting to opportunistically decompose (or rewrite) complex or error-prone operations into simpler ones.

DocETL is open-source and available on GitHub[2]. As of March 2025, it has already amassed **1.7k+ GitHub stars**, and has been used for pipelines ranging from domain-specific analysis (e.g., legal, climate science) to enterprise and personal productivity (e.g., analyzing customer support tickets, emails); over **400 users** have joined the corresponding Discord server.

Overall, finding optimal complex data processing pipelines is impossible given the infinite search space, non-determinism of LLMs, fuzziness of text, and ambiguity in task-specific success criteria. However, even in these difficult settings, DocETL is able to produce pipelines that are sufficiently accurate for practical needs, as is evidenced by our adoption across domains. DocETL is able to do so by leveraging the power of LLM agents in constrained ways, in

[2]`https://github.com/ucbepic/docetl`

conjunction with a powerful, but compact set of rewrite directives, decomposition into processing units that can be validated, as well as an opportunistic top-down exploration of the search space.

We make the following contributions in this paper:

(1) **Novel Rewrite Directives and Agent-Driven Rewriting:** We identify 13 new rewrite directives designed for LLM-based operators, addressing challenges unique to complex document processing. Unlike traditional rewrite rules, LLM agents are used to implement these directives. When a rule applies to a portion of a pipeline, agents synthesize appropriate prompts and parameters for new operations. For example, when decomposing a "summarize instances of misconduct" operation into multiple ones, an agent might create two steps: first, "list instances of misconduct given specific types (e.g., excessive force)," followed by "summarize each listed instance," crafting suitable prompts for each new operation.

(2) **Agent-Driven Plan Assessment:** We also use LLM agents to synthesize task-specific validation prompts for each operation, which are used to assess output quality. For instance, to verify a misconduct summary, an agent might create a prompt, "Does this summary include *all* instances of misconduct from the document?" Or, "Do all mentioned instances *actually exist* in the document?" The agents then execute plans on sample data and evaluate outputs using these custom prompts. This entire process happens without the user having to provide or manually validate examples.

(3) **Opportunistic Sub-plan Optimization:** Unlike traditional query optimizers that generate and evaluate a broad range of possible plans [6], we leverage an opportunistic top-down search strategy as shown in Figure 1: when we use a rewrite directive to decompose operators into new ones, we immediately optimize each new operator. We first check if each such operator is sufficiently accurate, based on the validation as described previously. If sufficiently accurate, we no longer optimize that operator, focusing instead on rewriting others. Thus, we opportunistically decompose (or apply rewrite directives to) operators that are not sufficiently accurate, Such an approach is necessary because enumerating and evaluating all theoretically-possible plans would be prohibitively time-consuming due to the inherent latencies in LLM operations.

We describe DocETL's programming model and operators in Section 2; our new LLM-centric rewrite directives in Section 3, the agentic optimizer that applies them, and evaluates the resulting plans, as well as the overall framework for optimization in Section 4. We present our initial evaluation in Section 5, where we demonstrate that across four unstructured document analysis tasks, DocETL finds plans that are **21 to 80% more accurate** than baselines. We discuss related work in Section 6.

2 DOCETL DSL AND OPERATORS

This section presents DocETL's programming model and operators.

2.1 Programming Model

DocETL processes collections of documents. A *document* comprises a set (or dictionary) of key (or equivalently, attribute)-value pairs, represented as a JSON object. For example, a police record could be a set of key-value pairs, where one key corresponds to the OCR output of the PDF, while other keys could capture metadata such as agency, file name, or creation date. A collection of documents or *dataset*, is a JSON array. This data representation lets us handle various data types and degrees of structure and easily reference data within operation prompts. Documents can be nested, e.g., a police record may contain an array of `related_documents` that each contain witness statements or evidence logs that are further nested.

DocETL DSL. DocETL employs YAML as its domain-specific language (DSL) to define data processing pipelines, for several reasons. First, YAML is flexible in accommodating complex multi-line prompts and examples, as well as output schemas and validation mechanisms, while intermixing formatting with arguments in Jinja [35]. Second, YAML is human-readable and doesn't require extensive coding expertise. Third, it is commonly used in industry for describing data pipelines (Apache Airflow, dbt, Prefect) and services (Kubernetes, Docker, Circle/Gitlab CI/CD). Finally, YAML serves as a simple intermediate format for representing the DocETL-optimized pipelines for human inspection, as well as for our no-code interface. That said, our optimization techniques are not dependent on YAML and are also applicable to other frameworks.

DocETL Pipelines. A DocETL *pipeline*, expressed in YAML, describes a sequence of *operations*. Each operation specifies its operator type, input source, prompt template, and output schema. The input source can be either the original dataset or the output of a previous operator. We refer to this input using pre-defined variables `input` or `inputs` depending on whether the input cardinality is one or many. The pipeline begins with dataset definitions, which serves as the initial input. As operators process data, they generate output obeying their schemas, which subsequent operators can then use. This structure allows for flexible and modular pipeline composition. DocETL supports a default model for the entire pipeline, with the option for per-operation model specifications.

Fault Tolerance. When executing an LLM-powered operator for many input documents in a pipeline, some operations may *occasionally* fail to adhere to the given prompt. While prior work assumes reliability in LLM outputs [1, 30, 38], DocETL explicitly addresses this variability: for each operator, users can specify validations as Python statements that evaluate to true or false, referencing document and output attributes. If any validation fails, the operation retries, using context from the failure to improve the likelihood of success in subsequent attempts.

2.2 LLM-Powered Operators

Here, we describe the LLM-powered operators in DocETL. Table 1 summarizes our operators; detailed syntax can be found in our documentation[3], and more thorough descriptions can be found in our tech report [43]. Most operators are LLM-versions of classic data processing operators, however, we introduce a new *resolve* operator, used to canonicalize variations in specific attribute values. In the following, for succinctness of description, we often conflate a *document*—a JSON object comprising key-value pairs and the basic unit of processing in a dataset with its *textual content*, typically a value for a specific key within the JSON object.

2.2.1 *Map*

The map operator applies an LLM-powered projection, also known as a *semantic projection*, to each document in the dataset. Let's consider an example of a map operation:

```yaml
- name: extract_officer_misconduct
  type: map
```

[3]https://www.docetl.org/

Table 1: DocETL's operator suite, divided into operators that leverage LLMs for semantic processing and auxiliary operators (*) that handle data manipulation. For each operator, we show the required user configuration and a high-level description of its functionality.

Operator	User Configuration	Description
Map	Prompt, output schema	Uses an LLM to execute a transformation per document, adding resulting new keys to the schema (and optionally omitting existing ones).
Parallel Map	Multiple prompts, output schemas	Uses an LLM to execute multiple independent transformations on each document in parallel, adding the new keys to the schema.
Reduce	Group-by keys, prompt, output schema	Uses an LLM to aggregate groups of documents sharing the same key values into one new document per distinct value.
Filter	Prompt returning boolean	Uses an LLM to evaluate a condition per document, retaining only those where the condition is true.
Resolve	Comparison prompt, resolution prompt	Uses an LLM to identify values for a given key(s) that fuzzily match across documents and generate canonical versions per group of values, replacing them in-place in the documents.
Equijoin	Comparison prompt	Uses an LLM to determine if pairs of documents from two datasets should be joined based on fuzzy/semantic matching of the corresponding keys.
Unnest*	Array/dict field to unnest	Flattens nested data structures by either creating separate documents from array elements or merging nested dictionary fields into parent documents.
Split*	Split key, chunk size	Divides documents into smaller chunks based on token count or other criteria, creating as many new docs as there are chunks.
Gather*	Context window configuration	Augments each chunk with context from surrounding chunks based on specified configuration (e.g., previous and next chunk counts), keeping the set of documents the same.

```
  output:
    schema:
      misconduct: "list[{officer_name: str, misconduct_instance: str}]"
  prompt: |
    Analyze the following police record:
    {{ input.document }}
    Extract any instances of officer misconduct or procedural violations. For
       each instance, provide the name of the officer involved and a brief
       description of the misconduct or violation.
```

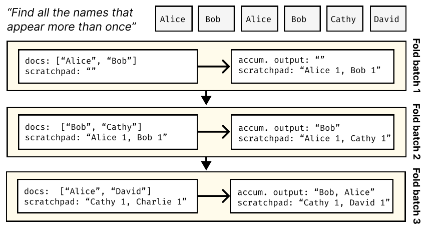
This operation processes each document independently, using the specified prompt. The output schema is a list of key-value pairs (of officer names and misconduct instances). This flexible, semi-structured output format allows for varying numbers of misconduct instances per document. DocETL supports prompts using Jinja2 templates, where "`{{ input.document }}`" allows for insertion of the current document's content. This functionality permits complex prompts with conditional logic (as we will see later). When applied, the map operation adds the new attributes specified in the output schema to the existing document. Users can override this behavior and return a subset of attributes by specifying a `drop_keys` list.

DocETL also supports *parallel* maps, where multiple independent transformations can be applied in parallel to each document. For example, one may extract misconduct while another summarizes relevant policies. Each operation enriches input documents with new attributes and can run in parallel rather than serially.

2.2.2 Reduce The reduce operator aggregates information across *multiple* documents based on a set of user-specified keys, ultimately producing *one* output document per unique combination of attribute values. For instance, for reducing police reports, the key set might include `officer_name` and `incident_date`, allowing for the grouping of all reports involving a specific officer on a particular date. Users can define prompt templates that access the grouped documents via `{{ inputs }}` (a list of documents sharing the same key values) and the specific key values for the current group via `{{ reduce_key }}`. By default, reduce operations are assumed to be associative, meaning that the order in which documents are processed does not affect the result. However, if the order is significant, users can specify `associative: False` in the operation definition.

A challenge arises when any given group of documents is too large for the LLM to correctly process. One could use folding or hierarchical merging to process the data in manageable batches [7, 17]. In folding, each input is serially processed, with an update to an accumulator (or aggregate), while hierarchical merging recursively aggregates inputs in a tree-like structure. DocETL currently implements a *batched folding* approach that starts with an empty accumulator and sequentially folds in batches of more than one document at a time. We chose folding because it permits non-associative reduce operations and maintains the original order of inputs. For example, when summarizing a textbook chapter, DocETL may chunk the text into sections, summarize each one, and then employ reduce to summarize the section summaries—a process that requires preserving the original reading order. DocETL automatically determines an optimal fold batch size when building the pipeline.

Figure 2: Reduce's iterative folding. Each batch takes several documents and the current scratchpad as input (left), and updates the mention counts in the scratchpad and accumulated output (right).


"Find all the names that appear more than once"
Alice
Bob
Alice
Bob
Cathy
David
docs: ["Alice", "Bob"]
scratchpad: ""
accum. output: ""
scratchpad: "Alice 1, Bob 1"
Fold batch 1
docs: ["Bob", "Cathy"]
scratchpad: "Alice 1, Bob 1"
accum. output: "Bob"
scratchpad: "Alice 1, Cathy 1"
Fold batch 2
docs: ["Alice", "David"]
scratchpad: "Cathy 1, Charlie 1"
accum. output: "Bob, Alice"
scratchpad: "Cathy 1, David 1"
Fold batch 3

To implement folding, users can provide (or DocETL can generate) a separate `fold_prompt`, which references the accumulated output and a batch of new inputs to fold into that output. We enhance the system prompt to allow the LLM to write extra notes to a scratchpad [34]—a technique that has been shown to improve accuracy by allowing it to maintain state. During each LLM call, we provide the current scratchpad along with the accumulated output and new inputs. The LLM returns both the updated accumulated output and scratchpad, which are passed to the next fold operation. Figure 2 depicts folding for a task to identify names of people mentioned more than once across documents. The scratchpad tracks all mentions of names. As each batch is processed, the LLM updates the scratchpad with new mentions and adds to the accumulated output any person now mentioned more than once.

2.2.3 Resolve This operator canonicalizes one or more keys across documents that represent slight variations of the same entity. Here, resolve reconciles small variations in officer names extracted as part of the map described in Section 2.2.1:

```
- name: resolve_officer_names
  type: resolve
  comparison_prompt: |
    Compare the following two officers from police records. Officer {{
       input1.officer_name }} mentioned in: {{ input1.record_txt }} and
       Officer {{ input2.officer_name }} mentioned in: {{ input2.record_txt
       }} Are these names referring to the same officer?
  resolution_prompt: |
    The following names correspond to the same officer:
    {% for input in inputs %}
      Name: {{ entry.officer_name }}
    {% endfor %}
    Provide an officer name (first and last) that best represents the matches.
  output:
    schema:
      officer_name: string
```

The user simply specifies how to detect variations, and how to canonicalize them. For instance, "`comparison_prompt`" checks whether two officer names are the same, while "`resolution_prompt`" chooses a canonical officer name from a list. DocETL then uses these prompts to compare and resolve the officer names. After this operation, the number of documents stays the same. The output schema specifies attributes to replace or add (if new) to each document. Resolve often follows `unnest` (Section 2.3.1), which flattens nested data structures. For example, in our police misconduct pipeline, after unnesting, each document would have distinct `officer_name` and `misconduct_instance` keys, allowing for name resolution across all mentions in the dataset. Note that users don't need to explicitly define the resolve operation in their pipeline; DocETL will automatically synthesize them if needed to ensure consistent entity references across the dataset. We will discuss how DocETL assesses the benefit of such rewrites in Section 4.1.

2.2.4 Other Operators While expressible using map and reduce, the following operators are added for convenience. We plan to add other operators (e.g., sort) in the future. **Filter** retains documents based on a condition specified in an LLM prompt, which uses a Jinja2 template referencing one or more document keys. **Equijoin** joins two datasets by comparing documents in pairs, using a `comparison_prompt` designed to elicit a binary answer from the LLM, referencing the documents as `left` and `right`. The equijoin operation doesn't require an output schema, as the left and right documents are merged to produce the results.

2.3 Auxiliary Operators

We present three essential operators that are *not* powered by LLMs, used as auxiliary steps to express complex tasks.

2.3.1 Unnest The unnest operator expands an array or dictionary into individual elements. For example, if a map extracts multiple officer names from police interrogation transcripts, each document may contain an array of names. To analyze officers individually across multiple interrogations, unnest creates a separate document for each officer name, effectively flattening the data. This operation can also elevate attributes from nested dictionaries, making them directly accessible for downstream processing.

2.3.2 Split The split operator divides long text into smaller chunks. It requires a split key (the text attribute), a split method (token or delimiter), and method-specific parameters (e.g., delimiter or chunk size). It generates unique identifiers and sequential numbers for each chunk to enable reassembly later in the pipeline. Resulting documents inherit the other attributes from the original documents.

2.3.3 Gather The gather operation complements the split operation by augmenting individual chunks with peripheral information necessary for understanding the chunk's content. Conceptually, gather is similar to windowing in SQL, as both allow ordered access to data beyond the current row or chunk, but gather is specifically designed for LLM-based processing. For example, in a transcript split into chunks, a chunk containing pronouns (e.g., "he" or "she") may lack speaker names, making it hard to understand. . Figure 3 demonstrates different ways to render chunks. The gather operation is highly flexible in rendering contextual information, allowing for the inclusion of full chunks (as in *(ii)*), portions of chunks (as in *(i)*), or transformations (e.g., summaries) of chunks (as in *(iii)*). Importantly, there may be map operations between the split and gather steps—allowing for the generation of additional context (such as summaries) that can be used to augment each chunk, before downstream processing. The output adds a new attribute to each input document, containing the rendered chunk with its peripheral context, with with special tags that demarcate what is the chunk and what is peripheral context.

Overall, in designing the DocETL DSL, we unified various single-document transformations (e.g., extraction, summarization) under `map` and `filter` operators, letting users express intent through prompts rather than learning multiple specialized operators. But for cross-document operations, we created distinct operators that capture specific processing patterns. For example, while `resolve` could theoretically be implemented using `equijoin`, `reduce`, and another `equijoin`, having a dedicated operator allows us to know that the user's intent is actually entity resolution, so we can better optimize the pipeline. Additionally, we distinguish `gather` from `reduce` because they serve different purposes: `reduce` performs many-to-one aggregation, whereas `gather` preserves cardinality while enriching documents with context—similar to SQL windowing functions.

3 REWRITE DIRECTIVES

We now introduce the *rewrite directives* that DocETL currently supports. We call these *directives* to indicate that they are abstract frameworks, with somewhat ambiguous semantics, that can be concretely instantiated by LLM agents in a multitude of ways, as opposed to *rules*, which are more concrete, complete, and robust. These directives are primarily designed to optimize the quality of outputs from DocETL pipelines through logical decomposition of individual operations. We focus on rewrite directives for map, reduce, and equijoin operators, with filter operators also supported through the application of map rewrite directives. We organize our rewrite directives into three main categories: data decomposition, projection synthesis, and LLM-centric improvements.

Throughout this section, we adopt the following notation: given operators A and B, we denote their composition as $A \rightarrow B$, where $(A \rightarrow B)(D) = B(A(D))$. For independent execution of operators, we use $A \parallel B$ to indicate that A and B are executed on the same input, independently. For readability, we may drop arguments—e.g., $\text{Map}_x(D)$ becomes Map_x. Similarly, we omit subscripts except when the same operator appears in multiple places. We further refer to the text content of the document, usually stored as one of the attributes, interchangeably with the document itself, for simplicity. The arrow $\Rightarrow$ denotes a (semantic) rewrite of the operator (or operator sequence) on the left into the form on the right.

3.1 Data Decomposition

Data decomposition is crucial when dealing with large documents, or when there are too many documents to fit in a prompt and get an accurate result for. We present two categories of rewrite directive here: *document chunking* and *multi-level aggregation*.

3.1.1 Document Chunking (Map) Large documents often exceed LLM context windows or effective reasoning capabilities, leading to incomplete or inconsistent results. Our primary rewrite directive for this case, which we call the *split directive*, is:

$$\text{Map}_x \Rightarrow \ ^{(2)}\ \text{Split} \xrightarrow{(3)} \text{Gather} \xrightarrow{(4)} \text{Map}_y \xrightarrow{(5)} \text{Reduce} \tag{1}$$

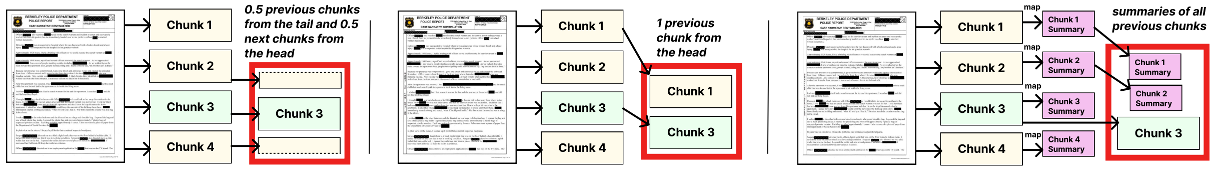


Figure 3: Split-Gather Pipeline: Illustration of processing a single long document. The split operation divides a long document into manageable chunks. The gather operation then augments each chunk with relevant context from peripheral chunks. The image demonstrates three different ways of rendering chunk 3 (i.e., three different gather configurations): *(i)* including fractional parts of surrounding chunks, *(ii)* including the full content of the first chunk, and *(iii)* including summaries of all previous chunks.

Ignoring the purple annotations, this directive rewrites map to: split the document into multiple chunks, gather peripheral context for each chunk, apply a modified map operation per chunk, and reduce the results. The prompt for Map_y may explicitly state that only a portion of the original document is being processed. To provide more flexibility and optimization opportunities, we introduce smaller decomposition directives, for steps (2)–(5) above:

$$\text{Split} \Rightarrow \text{Map} \rightarrow \text{Split} \tag{2}$$

$$\text{Split} \rightarrow \text{Gather} \Rightarrow \text{Split} \rightarrow (\text{Map}_s \parallel \text{Map}_h) \rightarrow \text{Gather} \tag{3}$$

$$\text{Gather} \Rightarrow \text{Gather} \rightarrow \text{Filter} \tag{4}$$

$$\text{Gather} \rightarrow \text{Map} \Rightarrow \text{Gather} \rightarrow \text{Map} \rightarrow \text{Unnest} \tag{5}$$

When splitting a document, three types of context prove particularly useful: document-level metadata, hierarchical information, and summaries of neighboring chunks. The smaller decomposition directives address these and other aspects of document processing:

- **Document-Level Metadata Extraction (2):** This directive introduces a map immediately prior to splitting, enabling the extraction of metadata relevant to *all* chunks. For example, when analyzing a legal contract, we might extract the contract date and parties involved from the first page, passing this information to every chunk to be rendered as part of a subsequent gather.
- **Header Lineage Context and Summarization (3):** This directive introduces two independent map operations: Map_h for extracting hierarchical information (e.g., headers), and Map_s for generating summaries of chunks. This allows us to provide each chunk with its relevant hierarchical context (e.g., parent headers for headers in a chunk) and/or a summary of preceding content.
- **Chunk Filtering (4):** Not all parts of a document may be relevant for processing. This directive introduces a filter step after gathering context, allowing us to exclude irrelevant chunks. This filter can be inferred; for instance, when processing a scientific paper, we might filter out acknowledgments or references sections if they're not pertinent to the analysis task; but they could still be used as context for other chunks if needed.
- **Flattening Nested Results (5):** When processing chunks with gathered context, map might produce nested results. This directive introduces an unnest operation to flatten these results, simplifying downstream processing. For example, if each chunk produces a list of extracted entities, unnesting would flatten these lists into a single collection of entities across all chunks.

3.1.2 Multi-Level Aggregation (Reduce) Large-scale aggregations can benefit from a hierarchical approach, aggregating data at a finer granularity before rolling up to the desired level. This decomposition is based on a semantic hierarchy in the data:

$$\text{Reduce}_{K,x} \Rightarrow \text{Reduce}_{K \cup K',y} \rightarrow \text{Reduce}_{K,z} \tag{6}$$

Here K is the reduce key, e.g., $K = \{\texttt{state}\}$, and K' represents additional keys for finer granularity, e.g., $K' = \{\texttt{city}\}$. y and z are LLM-powered aggregations for the sub-reduce and final reduce operations. For example, when summarizing voting patterns by state from social media posts, we might first aggregate data by state and city ($\text{Reduce}_{\{\texttt{state,city}\},y}$), then combine these city-level summaries to the state level ($\text{Reduce}_{\{\texttt{state}\},z}$). This approach can capture nuances that might be lost in a single, large-scale aggregation, and allows for intermediate validation.

3.2 LLM-Centric Improvements

This category addresses unique behaviors of LLMs that can be leveraged for optimization. We present two categories of rewrite directive: *gleaning* and *duplicate resolution*.

3.2.1 Gleaning (Map and Reduce) For this directive, we rely on the insight that when prompted with the previous inputs and outputs, and asked to improve the outputs, an LLM can iteratively refine the output. While iterative refinement has been implemented for knowledge graph entity extraction [10], we generalize this concept into a rewrite directive applicable to any map or reduce task. Our approach, which we call *gleaning*, employs separate data processing and validator LLM steps to iteratively improve output quality. We formalize the gleaning process for map operations as:

$$\text{Map} \Rightarrow \text{Map} \rightarrow (\text{Map}_v \rightarrow \text{Map}_i)^{\leq k} \tag{7}$$

Here, k represents the maximum number of refinement iterations, Map_v is a validation operation, and Map_i is a refinement operation. The process works as follows:

(1) Init: run the original map on the input document.
(2) Eval: separate validator (Map_v) checks output based on original prompt, init's output, and a task-specific validation prompt. The validator determines if refinement is needed and describes how to improve the output, if so.
(3) Refine: we use a refinement map (Map_i) to improve the previous iteration's output based on validator feedback. Importantly, this step retains the chat history, including the original prompt, its previous response, and the validator's feedback, so it can iteratively refine.
(4) Iterate: repeat up to k times, or no further refinement is needed.

A similar approach can be applied to reduce operations:

$$\text{Reduce} \Rightarrow \text{Reduce} \rightarrow (\text{Map}_v \rightarrow \text{Reduce}_i)^{\leq k} \tag{8}$$

For reduce operations, the refinement is applied at the level of a group, not to individual documents.

3.2.2 Duplicate Key Resolution (Reduce) A big challenge in LLM-powered data processing is that grouping, aggregation, and summarization is difficult due to the fact that LLM outputs are not

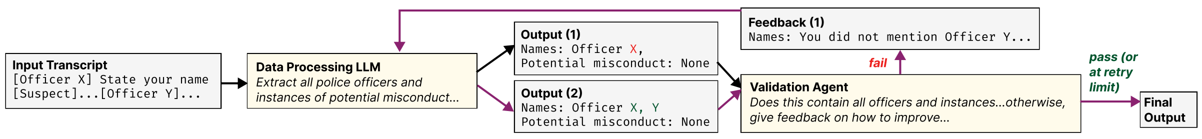

Input Transcript
[Officer X] State your name
[Suspect]...[Officer Y]...
Data Processing LLM
Extract all police officers and instances of potential misconduct...
Output (1)
Names: Officer X,
Potential misconduct: None
Output (2)
Names: Officer X, Y
Potential misconduct: None
Feedback (1)
Names: You did not mention Officer Y...
fail
Validation Agent
Does this contain all officers and instances...otherwise, give feedback on how to improve...
pass (or at retry limit)
Final Output

Figure 4: Gleaning process with $k = 1$ round of refinement. An LLM initially extracts information from an input transcript, and Officer Y is missing from the output. A validation agent (LLM-powered) identifies this omission and provides feedback. The original LLM incorporates this feedback in a second pass (shown with purple arrows), resulting in a more complete final output that includes both Officer X and Officer Y.

canonicalized, and may contain many semantic duplicates. To address semantic duplicates in reduce keys, especially those derived from LLM-powered operations, we introduce resolve operations:

$$\text{Reduce}_{K,x} \Rightarrow (\text{Resolve}_{k_1} \parallel \ldots \parallel \text{Resolve}_{k_m}) \rightarrow \text{Reduce}_{K,x} \quad (9)$$

Where $\{k_1, \ldots, k_m\} \subseteq K$ are each a disjoint subset of keys to be resolved. Each Resolve_{k_i} operation consolidates semantically equivalent values for the key k_i. We introduce this rewrite directive to address the inherent variability in LLM outputs: when LLMs are used to generate keys for reduce operations, they may produce semantically equivalent but syntactically different values. For example, "New York City," "NYC," and "The Big Apple" might all refer to the same entity. Without resolution, these would be treated as separate keys, leading to inaccurate aggregations.

3.3 Projection Synthesis

Projection synthesis strategies are inspired by projection pushdown optimizations in database systems. While selections (and selection pushdown) can also be synthesized, we did not implement this, as we found that agents are not very effective at determining whether certain data could be relevant to the query (they are overly biased by prompt wording and tend to be overly inclusive). Moreover, since an LLM-based selection is just as costly as a map, as both require an LLM call for every document, we focused on map operations that shrink the size of documents through a form of projection. We present several instances of projection synthesis directives:

$$\text{Map}_x \Rightarrow \text{Map}_{x_1} \rightarrow \text{Map}_{x_2} \rightarrow \cdots \rightarrow \text{Map}_{x_n} \quad (10)$$
$$\text{Map}_y \Rightarrow (\text{Map}_{y_1} \parallel \text{Map}_{y_2} \parallel \cdots \parallel \text{Map}_{y_m}) \rightarrow \text{Reduce} \quad (11)$$
$$\text{Reduce}_{K,x} \Rightarrow \text{Map}_y \rightarrow \text{Reduce}_{K,z} \quad (12)$$
$$\text{Equijoin}_x \Rightarrow (\text{Map}_{y,L} \parallel \text{Map}_{z,R}) \rightarrow \text{Equijoin}_w \quad (13)$$

- **Chaining** (10): This directive chains simpler projections for complex map operations, useful when a map prompt contains multiple instructions. Each Map_{x_i} builds on the previous result. For example, a legal document analysis could involve chained steps: extract clauses, summarize, and generate recommendations.
- **Isolating** (11): For map operations with *independent* subtasks, this directive splits them into separate projections to run in parallel, followed by a reduce step. For instance, customer feedback analysis could involve isolated projections to classify sentiment, identify features, and flag urgent issues.
- **Pre-Aggregation** (12): This directive filters and projects relevant data *from each document* before a reduce operation, improving both efficiency and the quality of the aggregation. For example, when summarizing shipping-related feedback by product category, each detailed review could first be projected into a concise summary of shipping comments, before aggregation.
- **Pre-Joining** (13): For complex equijoin operations, this directive preprocesses documents before joining. It is useful when direct comparison is computationally expensive—for example, matching research papers to funding opportunities could involve projecting papers to a short list of key themes and funding descriptions to criteria before joining.

One may wonder why each operator has its own directive (e.g., map before reduce, map before equijoin). This is because the criteria for applying the directive differ by operator. For example, in pre-joining, the LLM agent evaluates factors like the sufficiency of current keys and long/large attributes. If beneficial, it generates a prompt to create a new key-value pair for a more relevant data representation. Similarly, for other operators, the agent considers operator-specific factors to determine the directive's applicability.

4 OPTIMIZER

Here, we detail DocETL's query planning and optimization process. Users define their pipeline in a `pipeline.yaml` file, then run `docetl build pipeline.yaml` to generate a new YAML file with an optimized pipeline. DocETL's optimization involves two types of agents: *Generation agents*, which apply logical rewrite directives to create candidate plans (see "Apply Rewrites (Agent)" boxes in Figure 1), and *Validation agents*, which generate custom prompts to assess the quality of these plans. Per operation or sub-pipeline, validation agents evaluate candidate sub-plans on a data sample to select the optimal one, as shown by the green (selected) and gray (evaluated but not selected) sub-plans in Figure 1; we will describe both steps next. Our framework is reminiscent of top-down approaches like Cascades [16], but differs in its expansion criterion (using directives) and sub-plan evaluation via LLM-based validation. Unlike traditional cost-based optimizers, we focus on accuracy, with cost and latency constraints to be addressed in future work.

4.1 Optimization Approach

DocETL employs a top-down optimization approach that considers both individual operations and sub-pipelines, as visualized in Figure 1. We move from left to right, opting (recursively) to decompose any operations for which the accuracy is inadequate (as determined by the LLM validators). We summarize the process:

(1) **Pipeline Traversal and Sub-pipeline Identification**: We iterate through the pipeline from input to output (left to right). For each operation, we consider whether it, along with a suffix of the already-optimized operations to its left, forms a sub-pipeline that matches any rewrite directive. If no matching sub-pipeline is found, we treat the current operation as a single-operation sub-pipeline to optimize. For each identified sub-pipeline: *(i)* we use the validation agent to *synthesize a custom validation prompt* tailored to the specific task described by the sub-pipeline, and *(ii)* the validation agent *examines a sample of outputs* using this prompt to determine if there's room for improvement. If the agent concludes that the current implementation is satisfactory, we move on to the next operation without further optimization, as shown by the no-change

("NC") paths in Figure 1. Pseudocode can be found in our technical report [43].

(2) **Rewrite Directive Application and Recursive Optimization**: When optimization is needed, we apply matching rewrite directives to the sub-pipeline or individual operation. As illustrated in Figure 1, we explore rewrite directives from Section 3. For each applicable directive, an LLM agent synthesizes new operations and configurations (e.g., prompts, output schemas) to match the directive. On the creation of a new operation, we immediately optimize it, recursively, before continuing with the current optimization, as shown by the nested "Apply Rewrites" rectangles in the figure.

(3) **Plan Evaluation and Selection**: Multiple candidate plans can arise from the rewrite directives, as depicted by the various branches in Figure 1. We employ a two-stage evaluation process to select the best plan: First, we execute each plan on a sample of data and use the validation agent to rate the output for each document, computing an average rating per plan. We then select the top k rated plans (currently set to 6) for further comparison. Next, the agent performs pairwise comparisons between these top plans, evaluating their outputs against each other. The plan with the most "wins" in these comparisons is selected as the optimal plan for the current sub-pipeline or operation, represented by the green boxes in Figure 1. This hybrid approach balances efficiency and accuracy in plan evaluation, as pairwise comparisons are known to be ideal for assessing relative quality [32, 37], but with potentially 100+ candidate plans generated, comparing all pairs becomes computationally infeasible.

(4) **Pipeline Update**: We integrate the selected optimized plan into the pipeline, replacing the original operation or sub-pipeline.

To execute candidate plans (so we can compare their outputs), we sample data based on document size (larger documents have higher selection probability). As we optimize each sub-pipeline, we track its selectivity ratio (output documents / input documents) and use these ratios to adjust sample sizes for later operations. For example, if the first two operations have selectivities of 0.5 and 0.3, we increase the initial sample size by $(1/0.5/0.3) \approx 6.67$ when optimizing the third operation. This ensures sufficient data for optimization even after selective operations. However, sample documents may not fully represent the complete dataset; e.g., if the sampled documents fit within LLM context limits but some documents in the full dataset exceed them, we may encounter errors during full execution. We are developing methods to adapt plans accordingly during pipeline execution time.

4.2 Agent and System Implementation

Our generation agents apply rewrite directives to create diverse candidate plans, synthesizing appropriate configurations that encompass both *logical* choices (e.g., prompts, output schemas) and *physical* parameters (e.g., chunk sizes, batch sizes), akin to how traditional DBMSes maintain a logical-physical separation [15]. For physical parameter selection, where directly asking an LLM for optimal values (e.g., "what's the best chunk size for this document?") would be unreliable, our optimizer selects them empirically by generating candidate configurations, executing them on sampled data, and ranking results based on task-specific criteria. For chunk size determination in map operator decomposition, DocETL dynamically generates eight candidate chunk sizes: five based on percentages (15% to 75%, uniformly sampled) of the LLM's token limit, and six based on percentages (15% to 100%, uniformly sampled) of the average document length. For possible gather operations, DocETL evaluates multiple peripheral context strategies for each chunk size: *(1)* no peripheral context, *(2)* one previous chunk, *(3)* 1 previous and subsequent chunk, *(4)* number of previous chunks sized proportionally to the square root of the ratio of document size to chunk size, *(5)* 5 previous and 2 subsequent chunks for very small chunks (i.e., chunks <10% of document size), and *(6)* a summary of all previous chunks for small chunks (<20% of document size). Similarly, to determine fold batch size, DocETL empirically generates five candidate configurations at specific ratios of the model's maximum token limit (20%, 40%, 60%, 75%, and 90%). During optimization, our LLM agent generates two types of blocking rules to reduce unnecessary LLM comparisons when matching documents: embedding-based filtering, which only compares documents with cosine similarity above a threshold (tuned to recall 95% of true matches), and custom Python filters to eliminate obvious non-matches. Detailed strategies and empirical observations for each parameter selection approach are available in our technical report [43].

Our validation agents assess sub-pipeline effectiveness by synthesizing explicit validation criteria for each operator around accuracy, precision, and recall, rather than simply checking adherence to operation prompt instructions. Agents generate *multiple* criteria that evaluate different aspects of the output (e.g., for officer misconduct extraction, checking both supporting evidence and absence of hallucinations). By decomposing validation into specific and different testable properties, we enable more reliable evaluation [44, 46]. Moreover, agents evaluate outputs on a *sample* of data against these criteria to determine if further optimization is needed and compare plans. Our approach helps manage LLM validation uncertainties, while remaining practical for applications where traditional accuracy metrics and ground-truth may be undefined.

DocETL uses GPT-4o for optimization by default (though GPT-4o-mini is supported), while pipeline execution supports any LLM with tool calling capabilities. The system is implemented in Python (16K lines) with performance-critical components for resolve and equijoin execution, such as blocking rules, in Rust (2K lines).

5 EVALUATION

The primary goal in our evaluation is to show that DocETL's rewrite directives and optimization framework improves our ability to automatically analyze complex documents—all with no training labels or developer intervention needed. While finding optimal plans is impossible, we demonstrate that DocETL's approach of systematically decomposing tasks and documents to explore a search space of processing strategies yields plans that are sufficiently accurate.

Overall, we find that DocETL's plans yield 21% to 80% improvements in task-specific accuracy metrics such as precision, recall, and F1 score. We first consider three complex document processing tasks: legal contract analysis, declassified article analysis, and video game review analysis (Sections 5.1 to 5.3). These tasks represent different challenges: extracting structured information embedded within the semantic content of unstructured data, resolving entities and summarizing their information across documents, and reasoning about temporal consistency across long documents. For the legal contract analysis (Section 5.1), we compare against both recent LLM-powered systems (LOTUS [38], Palimpzest [30], and Aryn [1]) and traditional NLP baselines using spaCy [20] or NLTK [5]. For the video game review (Section 5.2) and declassified article (Section 5.3) tasks, we compare only against non-LLM baselines, as LOTUS, Palimpzest, and Aryn lack support for entity resolution and documents exceeding LLM context windows. For each task, our evaluation includes both task-specific metrics (customized variations of precision and recall) as well as a hallucination rate to measure factual consistency. Then, we evaluate DocETL on the challenging Biodex text classification task from Patel et al. [38], where our optimized pipeline achieves **33 to 80% improvements** in rank precision over baselines (Section 5.4). We conclude with case studies examining DocETL's application in real-world police misconduct identification, the effectiveness of LLM agent rewrites, and insights from user adoption (Section 5.5).

For all pipelines, we use the gpt-4o-mini model from OpenAI, and we run the experiments on a 2021 Macbook Pro with an M1 chip. The DocETL optimizer uses gpt-4o-mini, except in the Biodex task in Section 5.4, where we use gpt-4o. Additional implementation details can be found in our technical report [43]. All experiments were conducted in September 2024; Aryn and non-LLM baseline results were collected separately in February 2025. Note that all systems may have undergone significant changes since their respective evaluation dates.

5.1 Legal Contract Analysis

The Contract Understanding Atticus Dataset (CUAD) [19], includes 510 legal contracts with expert-labeled annotations across 41 categories of clauses, ranging from basic information (e.g., Document Name, Parties) to complex concepts (e.g., Most Favored Nation, IP Ownership, Post-Termination Services). The task is to extract text spans for each relevant clause type from each contract; not all contracts contain all types of clauses.

We evaluate on the first 50 contracts, comparing extractions against ground truth. An extraction is considered correct if *(i)* the clause type matches, and *(ii)* the extracted text span's Jaccard similarity with the ground truth span > 0.15. This threshold accommodates variation in LLM outputs while ensuring the model has correctly identified the clause's location; it is set fairly low because we provide no training examples, so the LLM does not know how much to extract—but large enough to ensure some match. We set other values for this and found the comparisons to be similar. We measure precision, recall, F1, and hallucination rate (proportion of extracted clauses not matching our 41 predefined categories).

5.1.1 Implementations We have five baselines:

(1) **DocETL Baseline:** Our unoptimized pipeline consists of a single map with a prompt to extract all relevant clauses, given one-sentence descriptions of the 41 clause types. The output schema specifies a list of objects with `clause_type` and `text_span` keys. The pipeline code is given in our technical report [43].

(2) **LOTUS Baseline:** We implement a pipeline using LOTUS's `sem_map` operator with the same prompt as DocETL's map operation, plus additional output structuring instructions since LOTUS does not support explicit output schema definitions.

(3) **Palimpzest Baseline:** We implement the extraction using Palimpzest's `convert` operator. In Palimpzest, rather than writing prompts directly, users provide schema descriptions from which the system generates prompts. We provided our clause type descriptions in the description of the schema.

(4) **Non-LLM Baseline:** We write a program, using the spaCy library [20], to loop over all clause types and extract the most semantically similar sentence (above a threshold of 0.9). We use spaCy's sentence splitter and embedding model, tok2vec.

(5) **Aryn Baseline:** We implement extraction using Aryn's `llm_query` operation with the same prompt as our LOTUS baseline, and the same output normalization procedure used with LOTUS to handle parsing errors and format inconsistencies.

(6) **DocETL's Optimized Plan:** DocETL's optimizer transforms the single map operation into an isolated projection decomposition with 21 independent map operations, each extracting 1-3 semantically related spans (e.g., grouping agreement and effective date extractions), followed by a reduce to combine all extracted clauses. Notably, the optimizer chose isolated projection (directive 11) over document chunking, suggesting that LLMs excel at focused extraction of small amounts of information even from lengthy documents.

Table 2: Legal Contract Analysis Results.

System	Avg Precision	Avg Recall	Avg F1	Avg # Chars	Avg Hallucination Rate
DocETL (Opt.)	0.401	**0.719**	**0.477**	162.60	**0.000**
DocETL (Unopt.)	0.341	0.430	0.379	49.35	0.072
LOTUS	0.402	0.471	0.393	46.301	0.073
Palimpzest	0.059	0.013	0.022	35.10	**0.000**
Aryn	**0.450**	0.370	0.352	49.56	0.069
Non-LLM	0.224	0.219	0.190	212.73	**0.000**

5.1.2 Results The results are shown in Table 2. DocETL's optimized plan performs significantly better than all baselines, achieving a **21.4% improvement in F1 over LOTUS**, the next best LLM-based plan, and a **67% improvement in recall** over the unoptimized DocETL pipeline—with no hallucinations. LOTUS, Aryn, and the unoptimized DocETL pipelines achieve similar scores and hallucination rates (6.9-7.3%). The non-LLM baseline achives much lower scores than the LLM-based methods, as well as longer text spans—because spans are forced to be at sentence-level granularity, which could be longer than necessary for short clauses like "document name" or "agreement date." Interestingly, Palimpzest's optimizer selected a code-based plan rather than an LLM-based one for this task—perhaps explaining its lower score.

While the optimized pipeline's cost and runtime are higher (Table 3), we prioritize accuracy, which often requires increased computational costs. The higher runtime and cost stems from the increased number of LLM calls in the new map operations, plus an additional reduce operation to combine their results. Further parallelism could help reduce the runtimes further, but this is not our focus. Costs will decrease as LLM pricing continues to fall—they have fallen by 1000× in 3 years, with a predicted drop of 10× per year [2]—and they become negligible when using open-source models. The optimization cost is only $1.58 (using gpt-4o-mini) and does not increase with dataset size, as it is done on a sample.

5.2 Game Review Analysis

We evaluate DocETL on temporal analysis of video game reviews from Steam (found at https://www.kaggle.com/datasets/najzeko/steam-reviews-2021). For each of 10 popular games, we create a

Table 3: Runtime and Cost Analysis for Legal Task. Palimpzest runtime is single-threaded and includes optimization time.

System	Runtime (s)	Cost ($)	Optimizer Cost ($)
DocETL (Opt.)	180.30	1.46	1.58
DocETL (Unopt.)	23.43	0.08	N/A
LOTUS	28.12	0.07	N/A
Palimpzest	84.07	Unknown*	Unknown*
Aryn	52.53	Unknown*	N/A
Non-LLM	217.99	0.00	N/A

*Costs are not reported by the system.

Table 4: Game Review Analysis Results

Metric	DocETL (Unopt.)	DocETL (Opt.)	Non-LLM
Hallucination Rate (lower is better)	0.465	**0.312**	N/A
Sentiment Accuracy (higher is better)	**0.664**	0.650	0.605
Kendall's Tau (higher is better)	0.470	**0.631**	N/A

document with 300 customer reviews with timestamps (but omit their ratings). Each document comprises concatenated reviews in no particular order, with lengths exceeding standard LLM context windows. The task is to identify 10 positive and 10 negative reviews per game, with their review IDs, and present these in chronological order. We evaluate the pipelines on: *(i)* hallucination rate, or the fraction of extracted review IDs that do not appear in the source, *(ii)* sentiment accuracy: whether the identified review sentiment matches the user's rating, computed only for non-hallucinated reviews, and *(iii)* Kendall's Tau correlation of the timestamp ordering, which measures how well the reviews are chronologically ordered.

5.2.1 Implementations Since the documents exceed context limits, we do not compare against existing LLM-based systems, which do not support documents beyond context windows. Our baseline DocETL pipeline consists of a single map to extract `positive_reviews` and `negative_reviews`, with documents truncated from the middle to fit the context window—effectively randomly sampling reviews. The exact pipeline can be found in our techical report [43]. The operation looks like the following:

DocETL's optimizer transforms this pipeline into: (a) A *split* operation that chunks input by token count (104,652 tokens per chunk): no gather operation (b) Two *map* operations per chunk—one each for positive/negative reviews—each incorporating one round of *gleaning* (directive 7) to ensure that the reviews are valid (c) A *reduce* operation to combine the positive and negative reviews from the chunks and present them in chronological order. We added a non-LLM baseline that extracts reviews via regex, classifies sentiment with NLTK and VADER [5, 21], and selects the first 10 positive and negative reviews. Since this baseline only performs classification (i.e., it is not a generative model), hallucination rate and Kendall's Tau metrics don't apply.

5.2.2 Results As shown in Table 4, we observe a **32.9% reduction in hallucination rate**, demonstrating more reliable review extraction. Sentiment accuracy remained stable (66.4% vs 65.0%), while Kendall's Tau improved by **34.3%**, indicating better temporal ordering. Both LLM-based approaches outperform the non-LLM baseline in sentiment accuracy, despite having to handle complex additional tasks beyond simple sentiment classification.

The optimized pipeline costs $1.48 (173.63s runtime) versus the baseline's $0.12 (29.27s). However, the baseline achieves this by truncating data to fit LLM context limits. With processing the full

Table 5: Declassified Article Analysis Results. Location metrics for baseline are N/A as its 233 distinct event types (mostly singleton categories) make meaningful location aggregation impossible.

Metric	DocETL (Unopt.)	DocETL (+Resolve Only)	DocETL (Opt.)	Non-LLM
Location Precision	N/A	0.994	**1.000**	0.6812
Location Recall	N/A	298	**435**	**435**
Distinct Event Types	164	83	83	N/A
Hallucination Rate	N/A	0.01	0.01	0.00

documents, the baseline would cost $0.28. This cost increase is justified by the improved temporal reasoning accuracy, and is due to steps like gleaning (which doubles operation cost); however, the gleaning validator consistently flagged temporal issues; with feedback like "The ... reviews are not sorted correctly by timestamp; they should be organized chronologically." The optimization cost was $6.60; however, this is a one-time cost. The non-LLM baseline had a runtime of 15.89 seconds.

5.3 Declassified Article Analysis

We evaluate DocETL's effectiveness on resolve and reduce tasks using 733 paranormal case files from The Black Vault, a repository of declassified international government documents, averaging 700 words each. Each article documents a reported paranormal event with details such as location and witness accounts. We scraped articles from their website and used Azure Document Intelligence to convert all PDF attachments to text. Our task is to determine the distinct locations for each type of paranormal event. The task involves two challenges: *(i)* standardizing event types across articles, and *(ii)* extracting and aggregating location mentions across articles for each event type.

We evaluated precision of extracted locations by first programatically verifying their presence in the source text and attempting to geocode them using the Nominatim API, based on OpenStreetMap. We also measured hallucination rate—a subset of precision—defined as the proportion of locations that don't exist in the source text. For locations in the text that could not be geocoded (e.g., specific rivers or mountain ranges), we performed manual verification.

5.3.1 Implementations We consider 4 pipelines. We only consider one LLM-powered baseline, written in DocETL, as other systems don't support resolve. This pipeline consists of: *(i)* a map to extract event type (e.g., "humanoid sighting") per article, and *(ii)* a reduce to collect distinct locations across all articles of each event type. DocETL's optimizer modified this pipeline in two ways. First, it synthesized a resolve between map and reduce to standardize event types (directive 9). Second, it optimized reduce by determining a fold batch size (41) to process document batches. To isolate the impact of the optimized reduce operation, we also evaluate the a version of this pipeline *(+resolve only)*, which uses the original reduce operation without batched folding. Our 4th pipeline represents a non-LLM baseline that extracts location (LOC) entities from article text using spaCy's `en_core_web_lg` model [20]. This script processes the resolved results from DocETL's optimized pipeline to establish a comparison point for location precision and recall.

5.3.2 Results As shown in Table 5, the baseline DocETL pipeline extracts 233 distinct event types with many semantic duplicates (e.g., "UFO Sighting", "Category: UFO Sighting", "Event Type: UFO Sighting"), making location aggregation impractical as most event types contain only one article. Adding resolve enables meaningful aggregation by consolidating to 83 event types. The +resolve only pipeline extracts 298 locations with 99.4% precision, and the optimized pipeline further improves this to 100% precision while extracting 435 locations **(46% higher recall)**. The non-LLM baseline matches the optimized pipeline's recall but with substantially lower precision (68.12% vs. 100%), highlighting the LLM's superior ability to *accurately* identify relevant locations in context. All systems exhibit low hallucination rates. This improved recall arises because batched folding allows the LLM to incrementally process and track distinct locations, rather than attempting to process all documents at once, where important details may be lost due to LLM context window overload [27, 31].

The resolve-only pipeline cost \$1.16 (307.36s), while the optimized version cost \$1.84 (\$1.34 + \$0.50 for optimization; 625.64s). The optimized pipeline's longer runtime results from multiple LLM calls per event type during folding, versus one call in the resolve-only version. The non-LLM baseline ran in 158.85s.

5.4 Biomedical Classification

We evaluate DocETL on the challenging Biodex biomedical drug reaction classification task from the LOTUS paper [38]. For each of 250 biomedical papers, the task involves identifying which of 24,300 adverse drug reactions (from the MedDRA list) are discussed. Performance is measured using rank-precision@k (RP@k), evaluating both accuracy and ranking of identified reactions. A higher score indicates that true positive reactions appear earlier in the list. We also evaluate the hallucination rate, measuring the proportion of identified reactions that are not present in the drug reaction list.

5.4.1 Implementations We compare against LOTUS using numbers from the first version of their preprint and our reimplementation of their pipeline (with gpt-4o-mini for LLM calls, text-embedding-3-small for embeddings). We implemented their best-performing join algorithm from the first version of their preprint, *map-search-filter*. See our technical report [43] for details.

In DocETL, we implement this task as an equijoin between articles and MedDRA labels, using a comparison prompt that asks "Can the following condition be found in the article?" We don't evaluate an unoptimized version due to the impractical number of LLM calls required (over 6 million). DocETL optimized this into a map-equijoin pipeline, where the map extracts medical conditions per article, and the equijoin uses synthesized blocking rules including an embedding similarity threshold of 0.5253 and a requirement that all words in the reaction label appear in the article text. Finally, we add a reduce operation that asks the LLM to rank the identified labels for each article from most to least confident, so we can measure ranking performance. We did not apply DocETL's reduce optimizations to this ranking step.

We also include a non-LLM baseline that identifies candidate labels by checking for exact substrings and ranks them by length. Given the large number of required comparisons, our non-LLM baseline uses keyword matching instead of more complex NLP libraries. Additional details can be found in our technical report [43].

5.4.2 Results At RP@25, which effectively measures recall since articles contain fewer than 25 labels in the ground truth, DocETL finds a plan with **80% improvement** over the baselines. For RP@5 and RP@10, DocETL shows **33% and 50% improvements**, respectively. The non-LLM baseline achieves lower RP@5 and RP@10 scores than the LLM-based methods, but a competitive RP@25. The improvement in recall may come from DocETL's synthesized blocking rules and the extra reduce step, which the DocETL pipeline necessarily includes to compute RP@k. In terms of hallucination rate, all systems perform well with essentially zero hallucinations. The difference between LOTUS' reported performance and our reimplementation may be attributed to model choices and prompting strategies, as we standardized on gpt-4o-mini without "few-shot" examples in the prompts.

Table 6: Biomedical Classification Results. Since most articles have fewer than 25 relevant labels, RP@25 effectively measures recall rather than ranking quality.

System	RP@5	RP@10	RP@25	Hallucination
DocETL	**0.281**	**0.313**	**0.371**	0.001
LOTUS (Our implementation of of map-search-filter)	0.213	0.207	0.206	0.000
LOTUS (Reported in Oct. 2024)	0.241	0.258	N/A	0.000
Non-LLM Baseline	0.106	0.158	0.262	0.000

Cost and Dataset Analysis. The non-LLM baseline takes 290.65 seconds to run. Our reimplemented LOTUS pipeline costs \$0.47 and takes 925 seconds to run. The DocETL pipeline costs \$3.65 and takes 463.28 seconds, with an additional optimization cost of \$2.37.

5.5 Case Studies, User Adoption, and Impact

To further evaluate our agentic optimizer, we conducted two case studies detailed in our technical report [43]: a real-world police misconduct identification application (as described in Example 1.1) and a stylized experiment on how effectively LLM agents instantiate rewrite directives. We summarize key findings from both studies, along with insights on user adoption and system limitations.

In our first case study, we built a pipeline to identify officer misconduct in ultra-long police records (>128K tokens) from the California Police Records Access Project (Example 1.1). DocETL's optimized pipeline **improved misconduct detection recall by 90%** compared to its unoptimized counterpart. For our second case study, we analyzed how LLMs transform abstract rewrite directives into concrete plans, examining 30 implementations across three directive types on legal contract analysis. As shown in Figure 5, despite significant variance in quality, many LLM-generated plans outperformed our baseline, with **47% achieving better precision** and **67% better recall**. 20% of LLM-generated plans had critical errors, like omitting document placeholders in prompts, leaving the LLM with no text to analyze. However, our optimizer was effective in weeding out bad plans, with our LLM-based evaluation mechanism correlating with F1-score (Kendall's tau of 0.642).

Since releasing DocETL as open source in October 2024, we've seen adoption across healthcare, legal, security, and scientific research domains, with users reporting significantly improved results "on the first try" for complex document tasks where other tools struggled. Our technical report [43] details additional use cases, post-release features, and discusses how DocETL differs from traditional database systems at every level of the system stack, ranging from physical and logical operators, to rewriting and optimization, to user specification and intent. We also discuss LLMs' non-deterministic effects on operator behavior and optimization—as

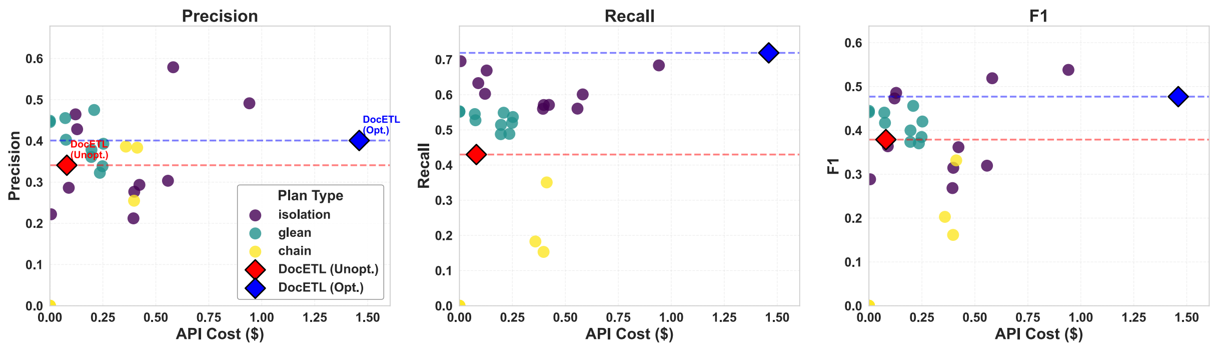


Figure 5: Cost vs. metrics (precision, recall, and F1) for 30 different LLM-generated implementations of rewrite directives applied to the legal contract analysis task. Each point represents a distinct plan implementation, colored by directive type; isolated projections (Equation (11), chaining projections (Equation (10), or gleaning (Equation (7)). The DocETL unoptimized baseline and optimized plan from Section 5.1 are shown with dashed lines for reference, though not generated in this experiment. Due to the optimizer's nondeterministic nature, some plans in this experiment achieved higher metrics than the original optimized plan.

well as our ongoing work to address current limitations through human-in-the-loop approaches.

6 RELATED WORK

LLM-powered data processing frameworks have recently gained significant attention in the database community. LOTUS [38] introduces *semantic operators*, defining a model for LLM-powered operations with accuracy guarantees with respect to high-quality LLM-powered reference algorithms. Palimpzest [30] provides a declarative framework focusing on map-like operations. Since our evaluation, Palimpzest has introduced a new optimizer with new results for the legal task [42], and LOTUS has a new join implementation based on model cascades, with improved results for the biomedical classification task. Aryn [1] offers a Spark-like API with PDF extraction capabilities and human-in-the-loop processing. Unlike DocETL, these systems primarily make simplifying assumptions about task complexity, typically focusing on extraction tasks or queries that capable LLMs can handle without decomposition. They employ various cost-based optimizations, including classical techniques like predicate pushdown [18] and ML-specific approaches like model cascades [24, 53]. However, when applied to complex document processing tasks, even state-of-the-art models fall short. DocETL addresses this limitation through agent-driven optimization, exploring decomposition to improve accuracy. Moreover, DocETL is the only system to support documents with lengths that exceed LLM context windows and introduces new operators (e.g., gather, split) for this, as well as for entity resolution as a first-class citizen.

Other LLM-powered data processing systems focus on different settings—typically making strong assumptions about the structure of the documents and predictability of format. ZenDB [29] optimizes SQL queries for templatized documents, while DocETL handles arbitrary documents. EVAPORATE [3] specializes in table extraction through code synthesis (only where applicable, in semi-structured settings), which could complement DocETL. Regarding LLM agents: Caesura [51] uses LLMs to translate natural language to SQL pipelines but leaves optimization for future work; CleanAgent [41] uses agents to standardize and clean data (and also does not consider optimization). Other systems propose specialized pipelines for specific tasks: for instance, Edge et al. [10] use a fixed map-reduce pipeline with predefined prompts for knowledge graph querying. A common limitation across these systems is inadequate context management, particularly for documents exceeding context windows or tasks requiring cross-document reasoning. While prompt optimization [26, 54] could complement DocETL, it falls short on complex document tasks, even with human guidance [55]. Moreover, LLMs have been leveraged for a variety of data tasks beyond document processing, such as join discovery [9, 25], database tuning [50], ML pipelines [45], natural language to SQL [40], semantic table understanding [8, 11]. and others [13], but not for complex document processing.

Finally, declarative frameworks for intelligent data processing have a rich history in database research through crowdsourcing systems like CrowdDB, Deco, CDB, and Qurk [14, 28, 33, 36]. While these systems use human rather than machine intelligence, they demonstrate declarative interfaces' power for complex tasks. DocETL extends this tradition to address the unique challenges of LLM-powered processing [37] through its flexible interface and agent-driven optimization.

7 CONCLUSION

We introduced DocETL, a declarative system that optimizes complex document processing tasks using LLMs. We introduced several novel rewrite directives, an agent-based framework for plan rewriting and evaluation, and an opportunistic optimization strategy. Our evaluation across four tasks demonstrated that DocETL can find plans with outputs 21-80% more accurate than baselines. DocETL is a first step toward an agentic optimizer for LLM-powered data processing. While exploring the large space of possible plan decompositions is hard, our approach shows that automated optimization is both feasible and beneficial. Future work will focus on reducing cost by considering cheaper models for simpler sub-tasks, and incorporating human feedback to refine plans.

ACKNOWLEDGMENTS

We acknowledge support from grants DGE-2243822, IIS-2129008, IIS-1940759, IIS-1940757, and IIS-2312991 awarded by the National Science Foundation, funds from the State of California, an NDSEG Fellowship, funds from the Alfred P. Sloan Foundation, as well as EPIC lab sponsors: Adobe, Google, G-Research, Microsoft, PromptQL, Sigma Computing, and Snowflake. Additional support was provided by Amazon and CAIT.

REFERENCES

[1] 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. 2024. The Design of an LLM-powered Unstructured Analytics System. arXiv:2409.00847 [cs.DB] https://arxiv.org/abs/2409.00847

[2] Guido Appenzeller. 2024. Welcome to LLMflation – LLM inference cost is going down fast. *a16z Blog, https://a16z.com/llmflation-llm-inference-cost/* (2024).

[3] 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. *arXiv preprint arXiv:2304.09433* (2023).

[4] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. 2023. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508* (2023).

[5] Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.".

[6] 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.

[7] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce online.. In *Nsdi*, Vol. 10. 20.

[8] Tianji Cong, Madelon Hulsebos, Zhenjie Sun, Paul Groth, and HV Jagadish. 2023. Observatory: Characterizing Embeddings of Relational Tables. *Proceedings of the VLDB Endowment* 17, 4 (2023), 849–862.

[9] Yuyang Dong, Chuan Xiao, Takuma Nozawa, Masafumi Enomoto, and Masafumi Oyamada. 2022. DeepJoin: Joinable Table Discovery with Pre-trained Language Models. *arXiv preprint arXiv:2212.07588* (2022).

[10] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. 2024. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130* (2024).

[11] Xi Fang, Weijie Xu, Fiona Anting Tan, Jiani Zhang, Ziqing Hu, Yanjun Qi, Scott Nickleach, Diego Socolinsky, Srinivasan Sengamedu, and Christos Faloutsos. 2024. Large Language Models on Tabular Data–A Survey. *arXiv preprint arXiv:2402.17944* (2024).

[12] Raul Castro Fernandez, Aaron J. Elmore, Michael J. Franklin, Sanjay Krishnan, and Chenhao Tan. 2023. How Large Language Models Will Disrupt Data Management. *Proc. VLDB Endow.* 16, 11 (jul 2023), 3302–3309. https://doi.org/10.14778/3611479.3611527

[13] Raul Castro Fernandez, Aaron J Elmore, Michael J Franklin, Sanjay Krishnan, and Chenhao Tan. 2023. How large language models will disrupt data management. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3302–3309.

[14] Michael J Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. 2011. CrowdDB: answering queries with crowdsourcing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 61–72.

[15] Goetz Graefe. 1993. Options in physical database design. *ACM Sigmod Record* 22, 3 (1993), 76–83.

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

[17] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. 1993. Maintaining views incrementally. *ACM SIGMOD Record* 22, 2 (1993), 157–166.

[18] Joseph M Hellerstein and Michael Stonebraker. 2005. Anatomy of a database system. *Readings in Database Systems,* (2005).

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

[20] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2020. *spaCy: Industrial-strength Natural Language Processing in Python.* https://doi.org/10.5281/zenodo.1212303

[21] Clayton Hutto and Eric Gilbert. 2014. Vader: A parsimonious rule-based model for sentiment analysis of social media text. In *Proceedings of the international AAAI conference on web and social media*, Vol. 8. 216–225.

[22] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. *arXiv preprint arXiv:2310.06839* (2023).

[23] Adam Tauman Kalai and Santosh S Vempala. 2024. Calibrated language models must hallucinate. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*. 160–171.

[24] 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).

[25] Moe Kayali, Anton Lykov, Ilias Fountalis, Nikolaos Vasiloglou, Dan Olteanu, and Dan Suciu. 2024. Chorus: Foundation Models for Unified Data Discovery and Exploration. *Proceedings of the VLDB Endowment* 17, 8 (2024), 2104–2114.

[26] 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*.

[27] Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same task, more tokens: the impact of input length on the reasoning performance of large language models. *arXiv preprint arXiv:2402.14848* (2024).

[28] Guoliang Li, Chengliang Chai, Ju Fan, Xueping Weng, Jian Li, Yudian Zheng, Yuanbing Li, Xiang Yu, Xiaohang Zhang, and Haitao Yuan. 2018. CDB: A crowd-powered database system. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1926–1929.

[29] Yiming Lin, Madelon Hulsebos, Ruiying Ma, Shreya Shankar, Sepanta Zeigham, Aditya G Parameswaran, and Eugene Wu. 2024. Towards Accurate and Efficient Document Analytics with Large Language Models. *arXiv preprint arXiv:2405.04674* (2024).

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

[31] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.

[32] Yinhong Liu, Han Zhou, Zhijiang Guo, Ehsan Shareghi, Ivan Vulić, Anna Korhonen, and Nigel Collier. 2024. Aligning with Human Judgement: The Role of Pairwise Preference in Large Language Model Evaluators. In *First Conference on Language Modeling*. https://openreview.net/forum?id=9gdZI7c6yr

[33] Adam Marcus, Eugene Wu, David R Karger, Samuel Madden, and Robert C Miller. 2011. Crowdsourced databases: Query processing with people. Cidr.

[34] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. 2021. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114* (2021).

[35] Pallets. 2024. Jinja. https://github.com/pallets/jinja/. Version 3.1.x.

[36] Aditya Ganesh Parameswaran, Hyunjung Park, Hector Garcia-Molina, Neoklis Polyzotis, and Jennifer Widom. 2012. Deco: declarative crowdsourcing. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. 1203–1212.

[37] Aditya G Parameswaran, Shreya Shankar, Parth Asawa, Naman Jain, and Yujie Wang. 2024. Revisiting Prompt Engineering via Declarative Crowdsourcing. *Cidr* (2024).

[38] Liana Patel, Siddharth Jha, Parth Asawa, Melissa Pan, Carlos Guestrin, and Matei Zaharia. 2024. Semantic Operators: A Declarative Model for Rich, AI-based Analytics Over Text Data. *arXiv preprint arXiv:2407.11418* (2024).

[39] Binghui Peng, Srini Narayanan, and Christos Papadimitriou. 2024. On limitations of the transformer architecture. *arXiv preprint arXiv:2402.08164* (2024).

[40] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2024. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *arXiv preprint arXiv:2410.01943* (2024).

[41] Danrui Qi and Jiannan Wang. 2024. CleanAgent: Automating Data Standardization with LLM-based Agents. *arXiv preprint arXiv:2403.08291* (2024).

[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] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G Parameswaran, and Eugene Wu. 2024. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. *arXiv preprint arXiv:2410.12189* (2024).

[44] Shreya Shankar, Haotian Li, Parth Asawa, Madelon Hulsebos, Yiming Lin, JD Zamfirescu-Pereira, Harrison Chase, Will Fu-Hinthorn, Aditya G Parameswaran, and Eugene Wu. 2024. spade: Synthesizing Data Quality Assertions for Large Language Model Pipelines. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4173–4186.

[45] Shreya Shankar and Aditya G Parameswaran. 2024. Building Reactive Large Language Model Pipelines with Motion. In *Companion of the 2024 International Conference on Management of Data*. 520–523.

[46] Shreya Shankar, JD Zamfirescu-Pereira, Björn Hartmann, Aditya Parameswaran, and Ian Arawjo. 2024. Who validates the validators? aligning llm-assisted evaluation of llm outputs with human preferences. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. 1–14.

[47] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*. PMLR, 31210–31227.

[48] Peiqi Sui, Eamon Duede, Sophie Wu, and Richard Jean So. 2024. Confabulation: The Surprising Value of Large Language Model Hallucinations. *arXiv preprint arXiv:2406.04175* (2024).
[49] Raphael Tang, Xinyu Zhang, Xueguang Ma, Jimmy Lin, and Ferhan Ture. 2023. Found in the middle: Permutation self-consistency improves listwise ranking in large language models. *arXiv preprint arXiv:2310.07712* (2023).
[50] Immanuel Trummer. 2022. DB-BERT: a Database Tuning Tool that" Reads the Manual". In *Proceedings of the 2022 international conference on management of data*. 190–203.
[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] Tempest A. van Schaik and Brittany Pugh. 2024. A Field Guide to Automatic Evaluation of LLM-Generated Summaries. In *Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. https://api.semanticscholar.org/CorpusID:271114432
[53] Xin Wang, Yujia Luo, Daniel Crankshaw, Alexey Tumanov, Fisher Yu, and Joseph E Gonzalez. 2017. Idk cascades: Fast deep learning by learning not to overthink. *arXiv preprint arXiv:1706.00885* (2017).
[54] Yuxin Wen, Neel Jain, John Kirchenbauer, Micah Goldblum, Jonas Geiping, and Tom Goldstein. 2024. Hard prompts made easy: Gradient-based discrete optimization for prompt tuning and discovery. *Advances in Neural Information Processing Systems* 36 (2024).
[55] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).
[56] Jun Zhao, Can Zu, Hao Xu, Yi Lu, Wei He, Yiwen Ding, Tao Gui, Qi Zhang, and Xuanjing Huang. 2024. LongAgent: Scaling Language Models to 128k Context through Multi-Agent Collaboration. *arXiv preprint arXiv:2402.11550* (2024).