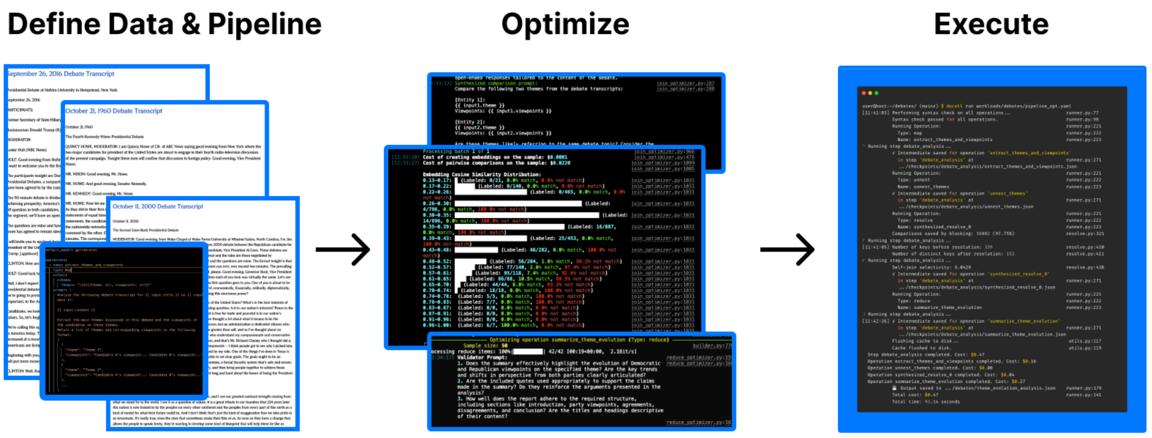
📜 DocETL: A System for Complex Document Processing

Stars 2.8k | Website docetl.org | Documentation docs | Discord 46 online | Paper arXiv


Define Data & Pipeline
Optimize
Execute
September 26, 2016 Debate Transcript
October 21, 1960 Debate Transcript
October 11, 2000 Debate Transcript

DocETL is a tool for creating and executing LLM-powered data processing pipelines. It offers a low-code, declarative YAML interface to define complex data operations on complex data.

When to Use DocETL

DocETL is the ideal choice when you're looking to **maximize correctness and output quality** for complex tasks over a collection of documents or unstructured datasets. You should consider using DocETL if:

- You have complex tasks that you want to represent via map-reduce (e.g., map over your documents, then group by the result of your map call & reduce)
- You're unsure how to best write your pipeline or sequence of operations to maximize LLM accuracy
- You're working with long documents that don't fit into a single prompt or are too lengthy for effective LLM reasoning
- You have validation criteria and want tasks to automatically retry when the validation fails

🚀 Features

- **Rich Suite of Operators**: Tailored for complex data processing, including specialized operators like "resolve" for entity resolution and "gather" for maintaining context when splitting documents.
- **Low-Code Interface**: Define your pipeline and prompts easily using YAML. You have 100% control over the prompts.
- **Flexible Processing**: Handle various document types and processing tasks across domains like law, medicine, and social sciences.
- **Accuracy Optimization**: Our optimizer leverages LLM agents to experiment with different logically-equivalent rewrites of your pipeline and automatically selects the most accurate version. This includes finding limits of how many documents to process in a single reduce operation before the accuracy plateaus.

⚡ Getting Started

To get started with DocETL:

1. Install the package (see installation for detailed instructions)
2. Define your pipeline in a YAML file. Want to use an LLM like ChatGPT or Claude to help you write your pipeline? See docetl.org/llms.txt for a big prompt you can copy paste into ChatGPT or Claude, before describing your task.
3. Run your pipeline using the DocETL command-line interface

🏛️ Project Origin

DocETL was created by members of the EPIC Data Lab and Data Systems and Foundations group at UC Berkeley. The EPIC (Effective Programming, Interaction, and Computation with Data) Lab focuses on developing low-code and no-code interfaces for data work, powered by next-generation predictive programming techniques. DocETL is one of the projects that emerged from our research efforts to streamline complex document processing tasks.

For more information about the labs and other projects, visit the EPIC Lab webpage and the Data Systems and Foundations webpage.

Installation

DocETL can be easily installed using pip, Python's package installer, or from source. Follow these steps to get DocETL up and running on your system:

🛠️ Prerequisites

Before installing DocETL, ensure you have Python 3.10 or later installed on your system. You can check your Python version by running:

📦 Installation via pip

1. Install DocETL using pip:

```
pip install docetl
```

If you want to use the parsing tools, you need to install the `parsing` extra:

```
pip install docetl[parsing]
```

This command will install DocETL along with its dependencies as specified in the pyproject.toml file. To verify that DocETL has been installed correctly, you can run the following command in your terminal:

```
docetl version
```

🔧 Installation from Source

To install DocETL from source, follow these steps:

1. Clone the repository:

```
git clone https://github.com/ucbepic/docetl.git
cd docetl
```

1. Install uv (if not already installed):

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

1. Install the project dependencies and DocETL:

```
uv sync --all-extras
```

If you want to use only the parsing extra:

```
uv sync --extra parsing
```

This will create a virtual environment and install all the required dependencies.

1. Set up your OpenAI API key:

Create a .env file in the project root and add your OpenAI API key:

```
OPENAI_API_KEY=your_api_key_here
```

Alternatively, you can set the OPENAI_API_KEY environment variable in your shell.

1. Run the basic test suite to ensure everything is working (this costs less than $0.01 with OpenAI):

```
make tests-basic
```

🚨 Troubleshooting

If you encounter any issues during installation, please ensure that:

- Your Python version is 3.10 or later
- You have the latest version of pip installed
- Your system meets all the requirements specified in the pyproject.toml file

For further assistance, please refer to the project's GitHub repository or reach out on the Discord server.

Tutorial: Analyzing Medical Transcripts with DocETL

This tutorial will guide you through the process of using DocETL to analyze medical transcripts and extract medication information. We'll create a pipeline that identifies medications, resolves similar names, and generates summaries of side effects and therapeutic uses.

Installation

First, let's install DocETL. Follow the instructions in the installation guide to set up DocETL on your system.

Setting up API Keys

DocETL uses LiteLLM under the hood, which supports various LLM providers. For this tutorial, we'll use OpenAI, as DocETL tests and existing pipelines are run with OpenAI.

Setting up your API Key

Set your OpenAI API key as an environment variable:

```
export OPENAI_API_KEY=your_api_key_here
```

Alternatively, you can create a `.env` file in your project directory and add the following line:

```
OPENAI_API_KEY=your_api_key_here
```

OpenAI Dependency

DocETL has been primarily tested with OpenAI's language models and relies heavily on their structured output capabilities. While we aim to support other providers in the future, using OpenAI is currently recommended for the best experience and most reliable results.

If you choose to use a different provider, be aware that you may encounter unexpected behavior or reduced functionality, especially with operations that depend on structured outputs. We use tool calling to extract structured outputs from the LLM's response, so make sure your provider supports tool calling.

If using a Gemini model, you can use the `gemini` prefix for the model name. For example, `gemini/gemini-2.0-flash`. (This has worked pretty well for us so far, and is so cheap!)

If using Ollama (e.g., llama 3.2), make sure your output schemas are not too complex, since these models are not as good as OpenAI for structured outputs! For example, use parallel map operations to reduce the number of output attributes per prompt.

Preparing the Data

Organize your medical transcript data in a JSON file as a list of objects. Each object should have a "src" key containing the transcript text. You can download the example dataset here.

Sample Data Structure

```
[
    {
        "src": "Doctor: Hello, Mrs. Johnson. How have you been feeling since starting the new medication, Lisinopril?\nPatient: Well, doctor, I've noticed my blood pressure has improved, but I've been experiencing some dry cough...",
    },
    {
        "src": "Doctor: Good morning, Mr. Smith. I see you're here for a follow-up on your Metformin prescription.\nPatient: Yes, doctor. I've been taking it regularly, but I'm concerned about some side effects I've been experiencing...",
    }
]
```

Save this file as `medical_transcripts.json` in your project directory.

Creating the Pipeline

Now, let's create a DocETL pipeline to analyze this data. We'll use a series of operations to extract and process the medication information:

1. **Medication Extraction**: Analyze each transcript to identify and list all mentioned medications.
2. **Unnesting**: The extracted medication list is flattened, such that each medication (and associated data) is a separate document. This operator is akin to the pandas `explode` operation.
3. **Medication Resolution**: Similar medication names are resolved to standardize the entries. This step helps in consolidating different variations or brand names of the same medication. For example, step 1 might extract "Ibuprofen" and "Motrin 800mg" as separate medications, and step 3 might resolve them to a single "Ibuprofen" entry.
4. **Summary Generation**: For each unique medication, generate a summary of side effects and therapeutic uses based on information from all relevant transcripts.

Create a file named `pipeline.yaml` with the following structure:

Pipeline Structure

```yaml
datasets:
  transcripts:
    path: medical_transcripts.json
    type: file

default_model: gpt-4o-mini

system_prompt: # This is optional, but recommended for better performance. It
is applied to all operations in the pipeline.
  dataset_description: a collection of transcripts of doctor visits
  persona: a medical practitioner analyzing patient symptoms and reactions to
medications

operations:
  - name: extract_medications
    type: map
    output:
      schema:
        medication: list[str]
    prompt: |
      Analyze the following transcript of a conversation between a doctor and a
patient:
      {{ input.src }}
      Extract and list all medications mentioned in the transcript.
      If no medications are mentioned, return an empty list.

  - name: unnest_medications
    type: unnest
    unnest_key: medication

  - name: resolve_medications
    type: resolve
    blocking_keys:
      - medication
    blocking_threshold: 0.6162
    comparison_prompt: |
      Compare the following two medication entries:
      Entry 1: {{ input1.medication }}
      Entry 2: {{ input2.medication }}
      Are these medications likely to be the same or closely related?
    embedding_model: text-embedding-3-small
    output:
      schema:
        medication: str
    resolution_prompt: |
      Given the following matched medication entries:
      {% for entry in inputs %}
      Entry {{ loop.index }}: {{ entry.medication }}
      {% endfor %}
      Determine the best resolved medication name for this group of entries.
The resolved
      name should be a standardized, widely recognized medication name that
best represents
      all matched entries.

  - name: summarize_prescriptions
    type: reduce
    reduce_key:
      - medication
    output:
      schema:
        side_effects: str
        uses: str
    prompt: |
      Here are some transcripts of conversations between a doctor and a
patient:

      {% for value in inputs %}
      Transcript {{ loop.index }}:
      {{ value.src }}
      {% endfor %}

      For the medication {{ reduce_key }}, please provide the following
information based on all the transcripts above:

      1. Side Effects: Summarize all mentioned side effects of {{ reduce_key
}}.
      2. Therapeutic Uses: Explain the medical conditions or symptoms for which
{{ reduce_key }} was prescribed or recommended.

      Ensure your summary:
      - Is based solely on information from the provided transcripts
      - Focuses only on {{ reduce_key }}, not other medications
      - Includes relevant details from all transcripts
      - Is clear and concise
      - Includes quotes from the transcripts

pipeline:
  steps:
    - name: medical_info_extraction
      input: transcripts
      operations:
        - extract_medications
        - unnest_medications
        - resolve_medications
        - summarize_prescriptions
  output:
    type: file
    path: medication_summaries.json
    intermediate_dir: intermediate_results
```

Running the Pipeline

Pipeline Performance

When running this pipeline on a sample dataset, we observed the following performance metrics using `gpt-4o-mini` as defined in the pipeline:

- Total cost: $0.10
- Total execution time: 49.13 seconds

If you want to run it on a smaller sample, set the `sample` parameter for the map operation. For example, `sample: 10` will run the pipeline on a random sample of 10 transcripts:

```
operations:
  - name: extract_medications
    type: map
    sample: 10
    ...
```

To execute the pipeline, run the following command in your terminal:

```
docetl run pipeline.yaml
```

This will process the medical transcripts, extract medication information, resolve similar medication names, and generate summaries of side effects and therapeutic uses for each medication. The results will be saved in `medication_summaries.json`.

Further Questions

What if I want to focus on a specific type of medication or medical condition?

You can modify the prompts in the `extract_medications` and `summarize_prescriptions` operations to focus on specific types of medications or medical conditions. For example, you could update the `extract_medications` prompt to only list medications related to cardiovascular diseases.

How can I improve the accuracy of medication name resolution?

The `resolve_medications` operation uses a blocking threshold and comparison prompt to identify similar medication names. Learn more about how to configure this operation in the resolve operation documentation. To automatically find the optimal blocking threshold for your data, you can invoke the optimizer, as described in the optimization documentation.

Can I process other types of medical documents with this pipeline?

Yes, you can adapt this pipeline to process other types of medical documents by modifying the input data format and adjusting the prompts in each operation. For example, you could use it to analyze discharge summaries, clinical notes, or research papers by updating the extraction and summarization prompts accordingly.

How can I optimize the performance of this pipeline?

If you're unsure about the optimal configuration for your specific use case, you can use DocETL's optimizer, which can be invoked using `docetl build` instead of `docetl run`. Learn more about the optimizer in the optimization documentation.

How can I use the pandas integration?

DocETL provides a pandas integration for a few operators (map, filter, merge, agg). Learn more about the pandas integration in the pandas documentation.

Tutorial: Analyzing Medical Transcripts with DocETL Python API

This tutorial will guide you through the process of using DocETL's Python API to analyze medical transcripts and extract medication information. We'll create a pipeline that identifies medications, resolves similar names, and generates summaries of side effects and therapeutic uses.

Prerequisites and Setup

For installation instructions, API key setup, and data preparation, please refer to the YAML-based tutorial. The prerequisite steps are identical for both the YAML and Python API approaches.

Common Setup

Both this Python API tutorial and the YAML-based tutorial share the same:

- Installation requirements
- API key configuration
- Data format and preparation steps

Once you've completed those steps from the main tutorial, you can continue with this Python API implementation.

Creating the Pipeline with Python API

Now, let's create a DocETL pipeline using the Python API to analyze this data. We'll use a series of operations to extract and process the medication information:

1. **Medication Extraction**: Analyze each transcript to identify and list all mentioned medications.
2. **Unnesting**: The extracted medication list is flattened, such that each medication (and associated data) is a separate document.
3. **Medication Resolution**: Similar medication names are resolved to standardize the entries. This step helps in consolidating different variations or brand names of the same medication.

4. **Summary Generation**: For each unique medication, generate a summary of side effects and therapeutic uses based on information from all relevant transcripts.

Create a Python script named `medical_analysis.py` with the following code:

```
from docetl.api import Pipeline, Dataset, MapOp, UnnestOp, ResolveOp,
ReduceOp, PipelineStep, PipelineOutput

# Define the dataset - JSON file with medical transcripts
dataset = Dataset(
    type="file",
    path="medical_transcripts.json"
)

# Define operations
operations = [
    # Extract medications from each transcript
    MapOp(
        name="extract_medications",
        type="map",
        prompt="""
        Analyze the following transcript of a conversation between a doctor
and a patient:
        {{ input.src }}
        Extract and list all medications mentioned in the transcript.
        If no medications are mentioned, return an empty list.
        """,
        output={
            "schema": {
                "medication": "list[str]"
            }
        }
    ),
    # Unnest to create separate items for each medication
    UnnestOp(
        name="unnest_medications",
        type="unnest",
        unnest_key="medication"
    ),
    # Resolve similar medication names
    ResolveOp(
        name="resolve_medications",
        type="resolve",
        blocking_keys=["medication"],
        blocking_threshold=0.6162,
        comparison_prompt="""
        Compare the following two medication entries:
        Entry 1: {{ input1.medication }}
        Entry 2: {{ input2.medication }}
        Are these medications likely to be the same or closely related?
        """,
        embedding_model="text-embedding-3-small",
        output={
            "schema": {
                "medication": "str"
            }
        },
        resolution_prompt="""
        Given the following matched medication entries:
        {% for entry in inputs %}
        Entry {{ loop.index }}: {{ entry.medication }}
        {% endfor %}
        Determine the best resolved medication name for this group of entries.
The resolved
        name should be a standardized, widely recognized medication name that
best represents
        all matched entries.
        """
    ),
    # Summarize side effects and uses for each medication
    ReduceOp(
        name="summarize_prescriptions",
        type="reduce",
        reduce_key=["medication"],
        prompt="""
        Here are some transcripts of conversations between a doctor and a
patient:

        {% for value in inputs %}
        Transcript {{ loop.index }}:
        {{ value.src }}
        {% endfor %}

        For the medication {{ reduce_key }}, please provide the following
information based on all the transcripts above:

        1. Side Effects: Summarize all mentioned side effects of {{ reduce_key
}}.
        2. Therapeutic Uses: Explain the medical conditions or symptoms for
which {{ reduce_key }} was prescribed or recommended.

        Ensure your summary:
        - Is based solely on information from the provided transcripts
        - Focuses only on {{ reduce_key }}, not other medications
        - Includes relevant details from all transcripts
        - Is clear and concise
        - Includes quotes from the transcripts
        """,
        output={
            "schema": {
                "side_effects": "str",
                "uses": "str"
            }
        }
    )
]

# Define pipeline step
step = PipelineStep(
    name="medical_info_extraction",
    input="transcripts",
    operations=[
        "extract_medications",
        "unnest_medications",
        "resolve_medications",
        "summarize_prescriptions"
    ]
)

# Define output
output = PipelineOutput(
    type="file",
    path="medication_summaries.json",
    intermediate_dir="intermediate_results"
)

# Define system prompt (optional but recommended)
system_prompt = {
    "dataset_description": "a collection of transcripts of doctor visits",
    "persona": "a medical practitioner analyzing patient symptoms and
reactions to medications"
}

# Create the pipeline
pipeline = Pipeline(
    name="medical_transcript_analysis",
    datasets={"transcripts": dataset},
    operations=operations,
    steps=[step],
    output=output,
    default_model="gpt-4o-mini",
    system_prompt=system_prompt
)

# Run the pipeline
cost = pipeline.run()
print(f"Pipeline execution completed. Total cost: ${cost:.2f}")
```

Running the Pipeline

To execute the pipeline, run the Python script:

```
python medical_analysis.py
```

This will process the medical transcripts, extract medication information, resolve similar medication names, and generate summaries of side effects and therapeutic uses for each medication. The results will be saved in `medication_summaries.json`.

Pipeline Performance

When running this pipeline on a sample dataset, we observed the following performance metrics using `gpt-4o-mini`:

- Total cost: $0.10
- Total execution time: 49.13 seconds

If you want to run it on a smaller sample, you can add a `sample` parameter to the `extract_medications` operation:

```
MapOp(
    name="extract_medications",
    type="map",
    sample=10,  # Process only 10 random transcripts
    # ... other parameters
)
```

Optimizing the Pipeline

If you want to optimize the pipeline configuration (such as finding the best blocking threshold for medication resolution):

```
# Create the pipeline
pipeline = Pipeline(
    # ... pipeline configuration as before
)

# Optimize the pipeline before running
optimized_pipeline = pipeline.optimize()

# Run the optimized pipeline
cost = optimized_pipeline.run()
print(f"Optimized pipeline execution completed. Total cost: ${cost:.2f}")
```

Further Questions

What if I want to focus on a specific type of medication or medical condition?

You can modify the prompts in the `extract_medications` and `summarize_prescriptions` operations to focus on specific types of medications or medical conditions. For example, you could update the `extract_medications` prompt to only list medications related to cardiovascular diseases.

How can I improve the accuracy of medication name resolution?

The `resolve_medications` operation uses a blocking threshold and comparison prompt to identify similar medication names. You can adjust the `blocking_threshold` parameter to control the sensitivity of the matching. Lower values will match more aggressively, while higher values require closer matches. You can also customize the comparison and resolution prompts.

Can I process other types of medical documents with this pipeline?

Yes, you can adapt this pipeline to process other types of medical documents by modifying the input data format and adjusting the prompts in each operation. For example, you could use it to analyze discharge summaries, clinical notes, or research papers by updating the extraction and summarization prompts accordingly.

How can I use the pandas integration?

DocETL provides a pandas integration for several operators (map, filter, merge, agg, split, gather, unnest). Here's an example of how to use it with the medication analysis:

```python
import pandas as pd
from docetl import SemanticAccessor

# Load data as DataFrame
df = pd.read_json("medical_transcripts.json")

# Use semantic map operation to extract medications
result_df = df.semantic.map(
    prompt="""
    Analyze the following transcript:
    {{ input.src }}
    Extract all medications mentioned.
    """,
    output_schema={"medications": "list[str]"}
)

# Continue processing with other pandas operations
```

Learn more about the pandas integration in the pandas documentation.

Best Practices for DocETL

This guide outlines best practices for using DocETL effectively, focusing on the most important aspects of pipeline creation, execution, and optimization.

Supported Models

DocETL supports many models through LiteLLM:

- OpenAI models (e.g., GPT-4, GPT-3.5-turbo)
- Anthropic models (e.g., Claude 2, Claude Instant)
- Google VertexAI models (e.g., chat-bison, text-bison)
- Cohere models
- Replicate models
- Azure OpenAI models
- Hugging Face models
- AWS Bedrock models (e.g., Claude, AI21, Cohere)
- Gemini models (e.g., gemini-1.5-pro)
- Ollama models (e.g., llama2)

For a complete and up-to-date list of supported models, please refer to the LiteLLM documentation. You can use the model name just like the litellm documentation (e.g., `openai/gpt-4o-mini` or `gemini/gemini-1.5-flash-002`).

While DocETL supports various models, it has been primarily tested with OpenAI's language models. Using OpenAI is currently recommended for the best experience and most reliable results, especially for operations that depend on structured outputs. We have also tried gemini-1.5-flash-002 and found it to be pretty good for a much cheaper price.

Pipeline Design

1. **Start Simple**: Begin with a basic pipeline and gradually add complexity as needed.

Example: Start with a simple extraction operation before adding resolution and summarization.

```yaml
operations:
  - name: extract_medications
    type: map
    output:
      schema:
        medication: list[str]
    prompt: |
      Extract and list all medications mentioned in the transcript:
      {{ input.src }}
```

1. **Modular Design**: Break down complex tasks into smaller, manageable operations.

Example: The medical transcripts pipeline in the tutorial demonstrates this by separating medication extraction, resolution, and summarization into distinct operations.

1. **Optimize Incrementally**: Optimize one operation at a time to ensure stability and verify improvements.

Example: After implementing the basic pipeline, you might optimize the `extract_medications` operation first:

```yaml
operations:
  - name: extract_medications
    type: map
    optimize: true
    output:
      schema:
        medication: list[str]
    prompt: |
      Extract and list all medications mentioned in the transcript:
      {{ input.src }}
```

Schema and Prompt Design

1. **Configure System Prompts**: Set up system prompts to provide context and establish the LLM's role for each operation. This helps generate more accurate and relevant responses.

Example:

```yaml
system_prompt:
  dataset_description: a collection of transcripts of doctor visits
  persona: a medical practitioner analyzing patient symptoms and reactions to
medications
```

The system prompt will be used as a system prompt for all operations in the pipeline.

1. **Keep Schemas Simple**: Use simple output schemas whenever possible. Complex nested structures can be difficult for LLMs to produce consistently.

Good Example (Simple Schema):

```
output:
  schema:
    medication: list[str] # Note that this is different from the example in
the tutorial.
```

Avoid (Complex Nested Structure):

```
output:
  schema:
    medications: "list[{name: str, dosage: {amount: float, unit: str,
frequency: str}}]"
```

1. **Clear and Concise Prompts**: Write clear, concise prompts for LLM operations, providing relevant context from input data. Instruct quantities (e.g., 2-3 insights, one summary) to guide the LLM.

Example: The `summarize_prescriptions` operation in the tutorial demonstrates a clear prompt with specific instructions:

```
prompt: |
  Here are some transcripts of conversations between a doctor and a patient:

  {% for value in inputs %}
  Transcript {{ loop.index }}:
  {{ value.src }}
  {% endfor %}

  For the medication {{ reduce_key }}, please provide the following
information based on all the transcripts above:

  1. Side Effects: Summarize all mentioned side effects of {{ reduce_key }}.
List 2-3 main side effects.
  2. Therapeutic Uses: Explain the medical conditions or symptoms for which {{
reduce_key }} was prescribed or recommended. Provide 1-2 primary uses.

  Ensure your summary:
  - Is based solely on information from the provided transcripts
  - Focuses only on {{ reduce_key }}, not other medications
  - Includes relevant details from all transcripts
  - Is clear and concise
  - Includes quotes from the transcripts
```

1. **Take advantage of Jinja Templating**: Use Jinja templating to dynamically generate prompts and provide context to the LLM. Feel free to use if statements, loops, and other Jinja features to customize prompts.

Example: Using Jinja conditionals and loops in a prompt (note that age is a made-up field for this example):

```
prompt: |
  Analyze the following medical transcript:
  {{ input.src }}

  {% if input.patient_age %}
  Note that the patient is {{ input.patient_age }} years old.
  {% endif %}

  Please extract the following information:
  {% for item in ["medications", "symptoms", "diagnoses"] %}
  - List all {{ item }} mentioned in the transcript
  {% endfor %}
```

1. **Validate Outputs**: Use the `validate` field to ensure the quality and correctness of processed data. This consists of Python statements that validate the output and optionally retry the LLM if one or more statements fail. To learn more about validation, see the validation documentation.

Example: Adding validation to the `extract_medications` operation:

```
operations:
  - name: extract_medications
    type: map
    output:
      schema:
        medication: list[str]
    prompt: |
      Extract and list all medications mentioned in the transcript:
      {{ input.src }}
    validate: |
      len(output.medication) > 0
      all(isinstance(med, str) for med in output.medication)
      all(len(med) > 1 for med in output.medication)
```

Handling Large Documents and Entity Resolution

1. **Chunk Large Inputs**: For documents exceeding token limits, consider using the optimizer to automatically chunk inputs.
2. **Use Resolve Operations**: Implement resolve operations before reduce operations when dealing with similar entities. Take care to write the compare prompts well to guide the LLM--often the optimizer-synthesized prompts are too generic.

Example: A more specific `resolve_medications` operation:

```
- name: resolve_medications
  type: resolve
  blocking_keys:
    - medication
  blocking_threshold: 0.6162
  comparison_prompt: |
    Compare the following two medication entries:
    Entry 1: {{ input1.medication }}
    Entry 2: {{ input2.medication }}

    Are these medications the same or closely related? Consider the following:
    1. Are they different brand names for the same active ingredient?
    2. Are they in the same drug class with similar effects?
    3. Are they commonly used as alternatives for the same condition?

    Respond with YES if they are the same or closely related, and NO if they
are distinct medications.
```

Optimization and Execution

1. **Use the Optimizer**: Leverage DocETL's optimizer for complex pipelines or when dealing with large documents.

Example: Run the optimizer on your pipeline:

```
docetl build pipeline.yaml
```

1. **Leverage Caching**: Take advantage of DocETL's caching mechanism to avoid redundant computations. DocETL caches by default.

To clear the cache:

```
docetl clear-cache
```

1. **Monitor Resource Usage**: Keep an eye on API costs and processing time, especially when optimizing. Use `gpt-4o-mini` for optimization (the default is `gpt-4o`) to save costs. Learn more about how to do this in the optimizer docs.

Additional Notes

- **Sampling Operations**: If you want to run an operation on a random sample of your data, you can set the `sample` parameter for that operation.

Example:

```
operations:
  - name: extract_medications
    type: map
    sample: 100
    output:
      schema:
        medication: list[str]
    prompt: |
      Extract and list all medications mentioned in the transcript:
      {{ input.src }}
```

- **Intermediate Output**: If you provide an intermediate directory in your configuration, the outputs of each operation will be saved to this directory. This allows you to inspect the results of individual steps in the pipeline and can be useful for debugging or analyzing the pipeline's progress.

Example:

```
pipeline:
  output:
    type: file
    path: medication_summaries.json
    intermediate_dir: intermediate_results
```

By following these comprehensive best practices and examples, you can create more efficient, reliable, and maintainable DocETL pipelines for your data processing tasks. Remember to iterate on your pipeline design, continuously refine your prompts, and leverage DocETL's optimization features to get the best results.