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.