Pointing to External Data and Custom Parsing

In DocETL, you have full control over your dataset JSONs. These JSONs typically contain objects with key-value pairs, where you can reference external files that you want to process in your pipeline. This referencing mechanism, which we call "pointing", allows DocETL to locate and process external files that require special handling before they can be used in your main pipeline.

Why Use Custom Parsing?

Consider these scenarios where custom parsing of referenced files is beneficial:

- Your dataset JSON references Excel spreadsheets containing sales data.
- You have entries pointing to scanned receipts in PDF format that need OCR processing.
- You want to extract text from Word documents or PowerPoint presentations by referencing their file locations.

In these cases, custom parsing enables you to transform your raw external data into a format that DocETL can process effectively within your pipeline. The pointing mechanism allows DocETL to locate these external files and apply custom parsing seamlessly. *(Pointing in DocETL refers to the practice of including references or paths to external files within your dataset JSON. Instead of embedding the entire content of these files, you simply "point" to their locations, allowing DocETL to access and process them as needed during the pipeline execution.)*

Dataset JSON Example

Let's look at a typical dataset JSON file that you might create:

```
[
  { "id": 1, "excel_path": "sales_data/january_sales.xlsx" },
  { "id": 2, "excel_path": "sales_data/february_sales.xlsx" }
]
```

In this example, you've specified paths to Excel files. DocETL will use these paths to locate and process the external files. However, without custom parsing, DocETL wouldn't know how to handle the contents of these files. This is where parsing tools come in handy.

Custom Parsing in Action

1. Configuration

To use custom parsing, you need to define parsing tools in your DocETL configuration file. Here's an example:

```
parsing_tools:
  - name: top_products_report
    function_code: |
      def top_products_report(document: Dict) -> List[Dict]:
          import pandas as pd

          # Read the Excel file
          filename = document["excel_path"]
          df = pd.read_excel(filename)

          # Calculate total sales
          total_sales = df['Sales'].sum()

          # Find top 500 products by sales
          top_products = df.groupby('Product')['Sales'].sum().nlargest(500)

          # Calculate month-over-month growth
          df['Date'] = pd.to_datetime(df['Date'])
          monthly_sales = df.groupby(df['Date'].dt.to_period('M'))
['Sales'].sum()
          mom_growth = monthly_sales.pct_change().fillna(0)

          # Prepare the analysis report
          report = [
              f"Total Sales: ${total_sales:,.2f}",
              "\nTop 500 Products by Sales:",
              top_products.to_string(),
              "\nMonth-over-Month Growth:",
              mom_growth.to_string()
          ]

          # Return a list of dicts representing the output
          # The input document will be merged into each output doc,
          # so we can access all original fields from the input doc.
          return [{"sales_analysis": "\n".join(report)}]

datasets:
  sales_reports:
    type: file
    source: local
    path: "sales_data/sales_paths.json"
    parsing:
      - function: top_products_report

  receipts:
    type: file
    source: local
    path: "receipts/receipt_paths.json"
```

```
    parsing:
      - input_key: pdf_path
        function: paddleocr_pdf_to_string
        output_key: receipt_text
        ocr_enabled: true
        lang: "en"
```

In this configuration:

- We define a custom `top_products_report` function for Excel files.
- We use the built-in `paddleocr_pdf_to_string` parser for PDF files.
- We apply these parsing tools to the external files referenced in the respective datasets.

2. Pipeline Integration

Once you've defined your parsing tools and datasets, you can use the processed data in your pipeline:

```
pipeline:
  steps:
    - name: process_sales
      input: sales_reports
      operations:
        - summarize_sales
    - name: process_receipts
      input: receipts
      operations:
        - extract_receipt_info
```

This pipeline will use the parsed data from both Excel files and PDFs for further processing.

How Data Gets Parsed and Formatted

When you run your DocETL pipeline, the parsing tools you've specified in your configuration file are applied to the external files referenced in your dataset JSONs. Here's what happens:

1. DocETL reads your dataset JSON file.
2. For each entry in the dataset, it looks at the parsing configuration you've specified.
3. It applies the appropriate parsing function to the file path provided in the dataset JSON.
4. The parsing function processes the file and returns the data in a format DocETL can work with (typically a list of strings).

Let's look at how this works for our earlier examples:

Excel Files (using top_products_report)

For an Excel file like "sales_data/january_sales.xlsx":

- The top_products_report function reads the Excel file.
- It processes the sales data and generates a report of top-selling products.
- The output might look like this:

```
Top Products Report - January 2023

1. Widget A - 1500 units sold
2. Gadget B - 1200 units sold
3. Gizmo C - 950 units sold
4. Doohickey D - 800 units sold
5. Thingamajig E - 650 units sold
   ...

Total Revenue: $245,000
Best Selling Category: Electronics
```

PDF Files (using paddleocr_pdf_to_string)

For a PDF file like "receipts/receipt001.pdf":

- The paddleocr_pdf_to_string function reads the PDF file.
- It uses PaddleOCR to perform optical character recognition on each page.
- The function combines the extracted text from all pages into a single string. The output might look like this:

```
RECEIPT
Store: Example Store
Date: 2023-05-15
Items:

1. Product A - $10.99
2. Product B - $15.50
3. Product C - $7.25
4. Product D - $22.00
   Subtotal: $55.74
   Tax (8%): $4.46
   Total: $60.20

Payment Method: Credit Card
Card Number: \***\* \*\*** \*\*\*\* 1234

Thank you for your purchase!
```

This parsed and formatted data is then passed to the respective operations in your pipeline for further processing.

Running the Pipeline

Once you've set up your pipeline configuration file with the appropriate parsing tools and dataset definitions, you can run your DocETL pipeline. Here's how:

1. Ensure you have DocETL installed in your environment.
2. Open a terminal or command prompt.
3. Navigate to the directory containing your pipeline configuration file.
4. Run the following command:

```
docetl run pipeline.yaml
```

Replace `pipeline.yaml` with the name of your pipeline file if it's different.

When you run this command:

1. DocETL reads your pipeline file.
2. It processes each dataset using the specified parsing tools.
3. The pipeline steps are executed in the order you defined.
4. Any operations you've specified (like `summarize_sales` or `extract_receipt_info`) are applied to the parsed data.
5. The results are saved according to your output configuration.

Built-in Parsing Tools

DocETL provides several built-in parsing tools to handle common file formats and data processing tasks. These tools can be used directly in your configuration by specifying their names in the `function` field of your parsing tools configuration. Here's an overview of the available built-in parsing tools:

Convert an Excel file to a string representation or a list of string representations.

Parameters:

Name	Type	Description	Default
`filename`	`str`	Path to the xlsx file.	*required*
`orientation`	`str`	Either "row" or "col" for cell arrangement.	`'col'`

Name	Type	Description	Default
`col_order`	`list[str] | None`	List of column names to specify the order.	`None`
`doc_per_sheet`	`bool`	If True, return a list of strings, one per sheet.	`False`

Returns:

Type	Description
`list[str]`	list[str]: String representation(s) of the Excel file content.

Source code in `docetl/parsing_tools.py`

```python
@with_input_output_key
def xlsx_to_string(
    filename: str,
    orientation: str = "col",
    col_order: list[str] | None = None,
    doc_per_sheet: bool = False,
) -> list[str]:
    """
    Convert an Excel file to a string representation or a list of string representations.

    Args:
        filename (str): Path to the xlsx file.
        orientation (str): Either "row" or "col" for cell arrangement.
        col_order (list[str] | None): List of column names to specify the order.
        doc_per_sheet (bool): If True, return a list of strings, one per sheet.

    Returns:
        list[str]: String representation(s) of the Excel file content.
    """
    import openpyxl

    wb = openpyxl.load_workbook(filename)

    def process_sheet(sheet):
        if col_order:
            headers = [
                col for col in col_order if col in sheet.iter_cols(1, sheet.max_column)
            ]
        else:
            headers = [cell.value for cell in sheet[1]]

        result = []
        if orientation == "col":
            for col_idx, header in enumerate(headers, start=1):
                column = sheet.cell(1, col_idx).column_letter
                column_values = [cell.value for cell in sheet[column][1:]]
                result.append(f"{header}: " + "\n".join(map(str, column_values)))
                result.append("")  # Empty line between columns
        else:  # row
            for row in sheet.iter_rows(min_row=2, values_only=True):
                row_dict = {
                    header: value for header, value in zip(headers, row) if header
                }
                result.append(
                    " | ".join(
                        [f"{header}: {value}" for header, value in row_dict.items()]
                    )
                )
```

```
            return "\n".join(result)

        if doc_per_sheet:
            return [process_sheet(sheet) for sheet in wb.worksheets]
        else:
            return [process_sheet(wb.active)]
```

options: show_root_heading: true heading_level: 3

Read the content of a text file and return it as a list of strings (only one element).

Parameters:

Name	Type	Description	Default
`filename`	`str`	Path to the txt or md file.	*required*

Returns:

Type	Description
`list[str]`	list[str]: Content of the file as a list of strings.

Source code in `docetl/parsing_tools.py`

```
156 @with_input_output_key
157 def txt_to_string(filename: str) -> list[str]:
158     """
159     Read the content of a text file and return it as a list of strings
160 (only one element).
161
162     Args:
163         filename (str): Path to the txt or md file.
164
165     Returns:
166         list[str]: Content of the file as a list of strings.
167     """
168     with open(filename, "r", encoding="utf-8") as file:
            return [file.read()]
```

options: show_root_heading: true heading_level: 3

Extract text from a Word document.

Parameters:

Name	Type	Description	Default
`filename`	`str`	Path to the docx file.	*required*

Returns:

Type	Description
`list[str]`	list[str]: Extracted text from the document.

Source code in `docetl/parsing_tools.py`

```python
@with_input_output_key
def docx_to_string(filename: str) -> list[str]:
    """
    Extract text from a Word document.

    Args:
        filename (str): Path to the docx file.

    Returns:
        list[str]: Extracted text from the document.
    """
    from docx import Document

    doc = Document(filename)
    return ["\n".join([paragraph.text for paragraph in doc.paragraphs])]
```

options: show_root_heading: true heading_level: 3

Transcribe speech from an audio file to text using Whisper model via litellm. If the file is larger than 25 MB, it's split into 10-minute chunks with 30-second overlap.

Parameters:

Name	Type	Description	Default
`filename`	`str`	Path to the mp3 or mp4 file.	*required*

Returns:

Type	Description
`list[str]`	list[str]: Transcribed text.

Source code in `docetl/parsing_tools.py`

```python
@with_input_output_key
def whisper_speech_to_text(filename: str) -> list[str]:
    """
    Transcribe speech from an audio file to text using Whisper model via litellm.
    If the file is larger than 25 MB, it's split into 10-minute chunks with 30-second overlap.

    Args:
        filename (str): Path to the mp3 or mp4 file.

    Returns:
        list[str]: Transcribed text.
    """

    from litellm import transcription

    file_size = os.path.getsize(filename)
    if file_size > 25 * 1024 * 1024:  # 25 MB in bytes
        from pydub import AudioSegment

        audio = AudioSegment.from_file(filename)
        chunk_length = 10 * 60 * 1000  # 10 minutes in milliseconds
        overlap = 30 * 1000  # 30 seconds in milliseconds

        chunks = []
        for i in range(0, len(audio), chunk_length - overlap):
            chunk = audio[i : i + chunk_length]
            chunks.append(chunk)

        transcriptions = []

        for i, chunk in enumerate(chunks):
            buffer = io.BytesIO()
            buffer.name = f"temp_chunk_{i}_{os.path.basename(filename)}"
            chunk.export(buffer, format="mp3")
            buffer.seek(0)  # Reset buffer position to the beginning

            response = transcription(model="whisper-1", file=buffer)
            transcriptions.append(response.text)

        return transcriptions
    else:
        with open(filename, "rb") as audio_file:
            response = transcription(model="whisper-1", file=audio_file)

        return [response.text]
```

options: show_root_heading: true heading_level: 3

Extract text from a PowerPoint presentation.

Parameters:

Name	Type	Description	Default
`filename`	`str`	Path to the pptx file.	*required*
`doc_per_slide`	`bool`	If True, return each slide as a separate document. If False, return the entire presentation as one document.	`False`

Returns:

Type	Description
`list[str]`	list[str]: Extracted text from the presentation. If doc_per_slide is True, each string in the list represents a single slide. Otherwise, the list contains a single string with all slides' content.

Source code in `docetl/parsing_tools.py`

```python
@with_input_output_key
def pptx_to_string(filename: str, doc_per_slide: bool = False) ->
list[str]:
    """
    Extract text from a PowerPoint presentation.

    Args:
        filename (str): Path to the pptx file.
        doc_per_slide (bool): If True, return each slide as a separate
            document. If False, return the entire presentation as one
document.

    Returns:
        list[str]: Extracted text from the presentation. If doc_per_slide
            is True, each string in the list represents a single slide.
            Otherwise, the list contains a single string with all slides'
            content.
    """
    from pptx import Presentation

    prs = Presentation(filename)
    result = []

    for slide in prs.slides:
        slide_content = []
        for shape in slide.shapes:
            if hasattr(shape, "text"):
                slide_content.append(shape.text)

        if doc_per_slide:
            result.append("\n".join(slide_content))
        else:
            result.extend(slide_content)

    if not doc_per_slide:
        result = ["\n".join(result)]

    return result
```

options: show_root_heading: true heading_level: 3

> Note to developers: We used this documentation as a reference.

This function uses Azure Document Intelligence to extract text from documents. To use this function, you need to set up an Azure Document Intelligence resource:

1. Create an Azure account if you don't have one
2. Set up a Document Intelligence resource in the Azure portal
3. Once created, find the resource's endpoint and key in the Azure portal
4. Set these as environment variables:

5. DOCUMENTINTELLIGENCE_API_KEY: Your Azure Document Intelligence API key
6. DOCUMENTINTELLIGENCE_ENDPOINT: Your Azure Document Intelligence endpoint URL

The function will use these credentials to authenticate with the Azure service. If the environment variables are not set, the function will raise a ValueError.

The Azure Document Intelligence client is then initialized with these credentials. It sends the document (either as a file or URL) to Azure for analysis. The service processes the document and returns structured information about its content.

This function then extracts the text content from the returned data, applying any specified formatting options (like including line numbers or font styles). The extracted text is returned as a list of strings, with each string representing either a page (if doc_per_page is True) or the entire document.

Parameters:

Name	Type	Description	Default
`filename`	`str`	Path to the file to be analyzed or URL of the document if use_url is True.	*required*
`use_url`	`bool`	If True, treat filename as a URL. Defaults to False.	`False`
`include_line_numbers`	`bool`	If True, include line numbers in the output. Defaults to False.	`False`
`include_handwritten`	`bool`	If True, include handwritten text in the output. Defaults to False.	`False`
`include_font_styles`	`bool`	If True, include font style information in the output. Defaults to False.	`False`
`include_selection_marks`	`bool`	If True, include selection marks in the output.	`False`

Name	Type	Description	Default
		Defaults to False.	
`doc_per_page`	`bool`	If True, return each page as a separate document. Defaults to False.	`False`

Returns:

Type	Description
`list[str]`	list[str]: Extracted text from the document. If doc_per_page is True, each string in the list represents a single page. Otherwise, the list contains a single string with all pages' content.

Raises:

Type	Description
`ValueError`	If DOCUMENTINTELLIGENCE_API_KEY or DOCUMENTINTELLIGENCE_ENDPOINT environment variables are not set.

Source code in `docetl/parsing_tools.py`

```python
@with_input_output_key
def azure_di_read(
    filename: str,
    use_url: bool = False,
    include_line_numbers: bool = False,
    include_handwritten: bool = False,
    include_font_styles: bool = False,
    include_selection_marks: bool = False,
    doc_per_page: bool = False,
) -> list[str]:
    """
    > Note to developers: We used [this documentation]
(https://learn.microsoft.com/en-us/azure/ai-services/document-
intelligence/how-to-guides/use-sdk-rest-api?view=doc-intel-
4.0.0&tabs=windows&pivots=programming-language-python) as a reference.

    This function uses Azure Document Intelligence to extract text from
documents.
    To use this function, you need to set up an Azure Document
Intelligence resource:

    1. [Create an Azure account](https://azure.microsoft.com/) if you
don't have one
    2. Set up a Document Intelligence resource in the [Azure portal]
(https://portal.azure.com/#create/Microsoft.CognitiveServicesFormRecogniz
er)
    3. Once created, find the resource's endpoint and key in the Azure
portal
    4. Set these as environment variables:
       - DOCUMENTINTELLIGENCE_API_KEY: Your Azure Document Intelligence
API key
       - DOCUMENTINTELLIGENCE_ENDPOINT: Your Azure Document Intelligence
endpoint URL

    The function will use these credentials to authenticate with the
Azure service.
    If the environment variables are not set, the function will raise a
ValueError.

    The Azure Document Intelligence client is then initialized with these
credentials.
    It sends the document (either as a file or URL) to Azure for
analysis.
    The service processes the document and returns structured information
about its content.

    This function then extracts the text content from the returned data,
    applying any specified formatting options (like including line
numbers or font styles).
    The extracted text is returned as a list of strings, with each string
    representing either a page (if doc_per_page is True) or the entire
document.

    Args:
        filename (str): Path to the file to be analyzed or URL of the
document if use_url is True.
        use_url (bool, optional): If True, treat filename as a URL.
```

```
Defaults to False.
        include_line_numbers (bool, optional): If True, include line
numbers in the output. Defaults to False.
        include_handwritten (bool, optional): If True, include
handwritten text in the output. Defaults to False.
        include_font_styles (bool, optional): If True, include font style
information in the output. Defaults to False.
        include_selection_marks (bool, optional): If True, include
selection marks in the output. Defaults to False.
        doc_per_page (bool, optional): If True, return each page as a
separate document. Defaults to False.

    Returns:
        list[str]: Extracted text from the document. If doc_per_page is
True, each string in the list represents
                   a single page. Otherwise, the list contains a single
string with all pages' content.

    Raises:
        ValueError: If DOCUMENTINTELLIGENCE_API_KEY or
DOCUMENTINTELLIGENCE_ENDPOINT environment variables are not set.
    """

    from azure.ai.documentintelligence import DocumentIntelligenceClient
    from azure.ai.documentintelligence.models import
AnalyzeDocumentRequest
    from azure.core.credentials import AzureKeyCredential

    key = os.getenv("DOCUMENTINTELLIGENCE_API_KEY")
    endpoint = os.getenv("DOCUMENTINTELLIGENCE_ENDPOINT")

    if key is None:
        raise ValueError("DOCUMENTINTELLIGENCE_API_KEY environment
variable is not set")
    if endpoint is None:
        raise ValueError(
            "DOCUMENTINTELLIGENCE_ENDPOINT environment variable is not
set"
        )

    document_analysis_client = DocumentIntelligenceClient(
        endpoint=endpoint, credential=AzureKeyCredential(key)
    )

    if use_url:
        poller = document_analysis_client.begin_analyze_document(
            "prebuilt-read", AnalyzeDocumentRequest(url_source=filename)
        )
    else:
        with open(filename, "rb") as f:
            poller =
document_analysis_client.begin_analyze_document("prebuilt-read", f)

    result = poller.result()

    style_content = []
    content = []

    if result.styles:
        for style in result.styles:
            if style.is_handwritten and include_handwritten:
```

```
                handwritten_text = ",".join(
                    [
                        result.content[span.offset : span.offset +
span.length]
                        for span in style.spans
                    ]
                )
                style_content.append(f"Handwritten content:
{handwritten_text}")

            if style.font_style and include_font_styles:
                styled_text = ",".join(
                    [
                        result.content[span.offset : span.offset +
span.length]
                        for span in style.spans
                    ]
                )
                style_content.append(f"'{style.font_style}' font style:
{styled_text}")

        for page in result.pages:
            page_content = []

            if page.lines:
                for line_idx, line in enumerate(page.lines):
                    if include_line_numbers:
                        page_content.append(f" Line #{line_idx}:
{line.content}")
                    else:
                        page_content.append(f"{line.content}")

            if page.selection_marks and include_selection_marks:
                # TODO: figure this out
                for selection_mark_idx, selection_mark in
enumerate(page.selection_marks):
                    page_content.append(
                        f"Selection mark #{selection_mark_idx}: State is
'{selection_mark.state}' within bounding polygon "
                        f"'{selection_mark.polygon}' and has a confidence of
{selection_mark.confidence}"
                    )

            content.append("\n".join(page_content))

        if doc_per_page:
            return style_content + content
        else:
            return [
                "\n\n".join(
                    [
                        "\n".join(style_content),
                        "\n\n".join(
                            f"Page {i+1}:\n{page_content}"
                            for i, page_content in enumerate(content)
                        ),
                    ]
                )
            ]
```

options: heading_level: 3 show_root_heading: true

Extract text and image information from a PDF file using PaddleOCR for image-based PDFs.

Note: this is very slow!!

Parameters:

Name	Type	Description	Default
`input_path`	`str`	Path to the input PDF file.	*required*
`doc_per_page`	`bool`	If True, return a list of strings, one per page. If False, return a single string.	`False`
`ocr_enabled`	`bool`	Whether to enable OCR for image-based PDFs.	`True`
`lang`	`str`	Language of the PDF file.	`'en'`

Returns:

Type	Description
`list[str]`	list[str]: Extracted content as a list of formatted strings.

Source code in `docetl/parsing_tools.py`

```python
@with_input_output_key
def paddleocr_pdf_to_string(
    input_path: str,
    doc_per_page: bool = False,
    ocr_enabled: bool = True,
    lang: str = "en",
) -> list[str]:
    """
    Extract text and image information from a PDF file using PaddleOCR for image-based PDFs.

    **Note: this is very slow!!**

    Args:
        input_path (str): Path to the input PDF file.
        doc_per_page (bool): If True, return a list of strings, one per page.
            If False, return a single string.
        ocr_enabled (bool): Whether to enable OCR for image-based PDFs.
        lang (str): Language of the PDF file.

    Returns:
        list[str]: Extracted content as a list of formatted strings.
    """
    import fitz
    import numpy as np
    from paddleocr import PaddleOCR

    ocr = PaddleOCR(use_angle_cls=True, lang=lang)

    pdf_content = []

    with fitz.open(input_path) as pdf:
        for page_num in range(len(pdf)):
            page = pdf[page_num]
            text = page.get_text()
            images = []

            # Extract image information
            for img_index, img in enumerate(page.get_images(full=True)):
                rect = page.get_image_bbox(img)
                images.append(f"Image {img_index + 1}: bbox {rect}")

            page_content = f"Page {page_num + 1}:\n"
            page_content += f"Text:\n{text}\n"
            page_content += "Images:\n" + "\n".join(images) + "\n"

            if not text and ocr_enabled:
                mat = fitz.Matrix(2, 2)
                pix = page.get_pixmap(matrix=mat)
                img = np.frombuffer(pix.samples, dtype=np.uint8).reshape(
                    pix.height, pix.width, 3
                )

                ocr_result = ocr.ocr(img, cls=True)
                page_content += "OCR Results:\n"
                for line in ocr_result[0]:
```

```
                    bbox, (text, _) = line
                    page_content += f"{bbox}, {text}\n"

            pdf_content.append(page_content)

    if not doc_per_page:
        return ["\n\n".join(pdf_content)]

    return pdf_content
```

options: heading_level: 3 show_root_heading: true

Using Function Arguments with Parsing Tools

When using parsing tools in your DocETL configuration, you can pass additional arguments to the parsing functions.

For example, when using the xlsx_to_string parsing tool, you can specify options like the orientation of the data, the order of columns, or whether to process each sheet separately. Here's an example of how to use such kwargs in your configuration:

```yaml
datasets:
  my_sales:
    type: file
    source: local
    path: "sales_data/sales_paths.json"
    parsing_tools:
      - name: excel_parser
        function: xlsx_to_string
        orientation: row
        col_order: ["Date", "Product", "Quantity", "Price"]
        doc_per_sheet: true
```

Contributing Built-in Parsing Tools

While DocETL provides several built-in parsing tools, the community can always benefit from additional utilities. If you've developed a parsing tool that you think could be useful for others, consider contributing it to the DocETL repository. Here's how you can add new built-in parsing utilities:

1. Fork the DocETL repository on GitHub.
2. Clone your forked repository to your local machine.
3. Navigate to the `docetl/parsing_tools.py` file.
4. Add your new parsing function to this file. The function should also be added to the `PARSING_TOOLS` dictionary.
5. Update the documentation in the function's docstring.

6. Create a pull request to merge your changes into the main DocETL repository.

Guidelines for Contributing Parsing Tools

When contributing a new parsing tool, make sure it follows these guidelines:

- The function should have a clear, descriptive name.
- Include comprehensive docstrings explaining the function's purpose, parameters, and return value. The return value should be a list of strings.
- Handle potential errors gracefully and provide informative error messages.
- If your parser requires additional dependencies, make sure to mention them in the pull request.

Creating Custom Parsing Tools

If the built-in tools don't meet your needs, you can create your own custom parsing tools. Here's how:

1. Define your parsing function in the `parsing_tools` section of your configuration.
2. Ensure your function takes a item (dict) as input and returns a list of items (dicts).
3. Use your custom parser in the `parsing` section of your dataset configuration.

For example:

```yaml
parsing_tools:
  - name: my_custom_parser
    function_code: |
      def my_custom_parser(item: Dict) -> List[Dict]:
          # Your custom parsing logic here
          return [processed_data]

datasets:
  my_dataset:
    type: file
    source: local
    path: "data/paths.json"
    parsing:
      - function: my_custom_parser
```