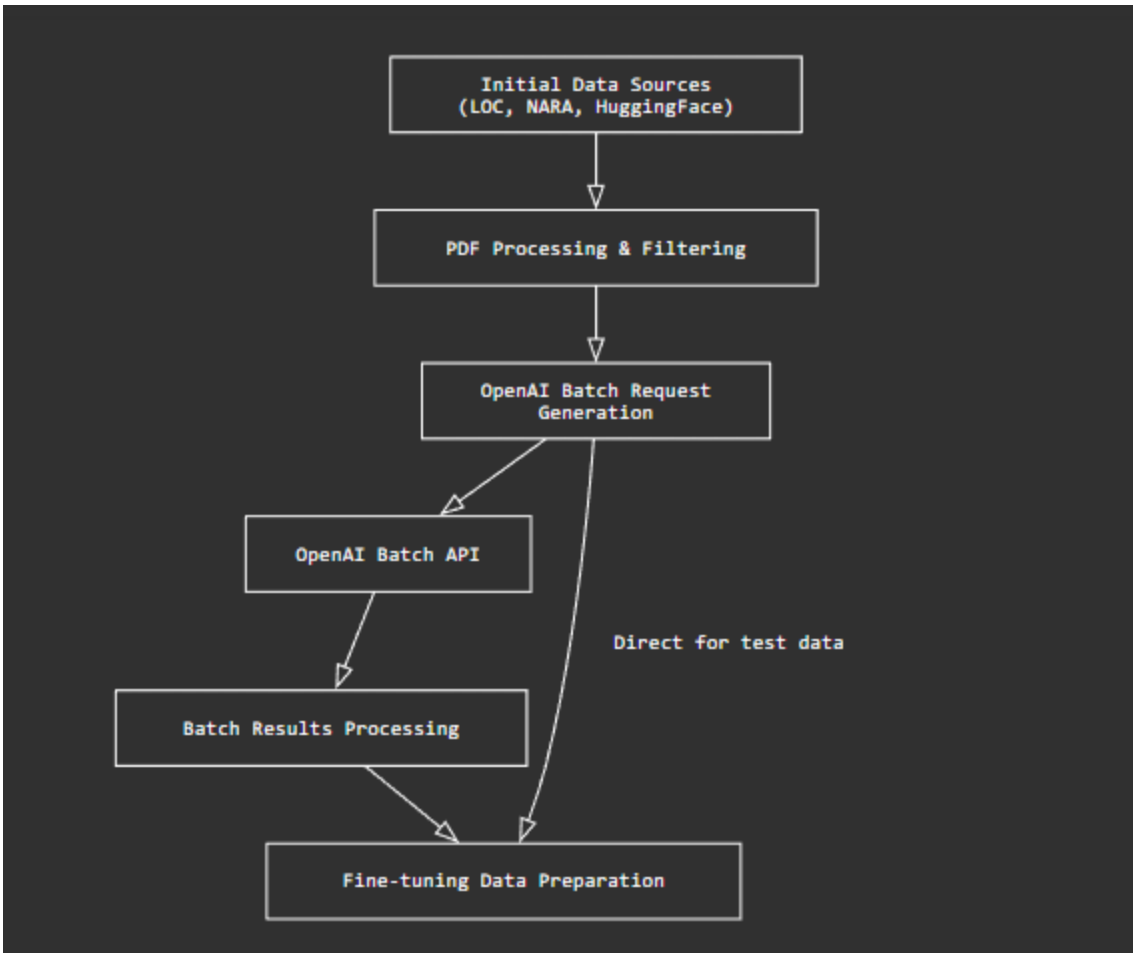


The **olmOCR** project includes functionality for managing, processing, and preparing various datasets for training, as well as handling batch results from external services. This encompasses downloading, converting, cleaning, and structuring document data, such as PDFs, images, and transcriptions, into **olmOCR**-compatible formats, including Markdown and single-page PDFs.

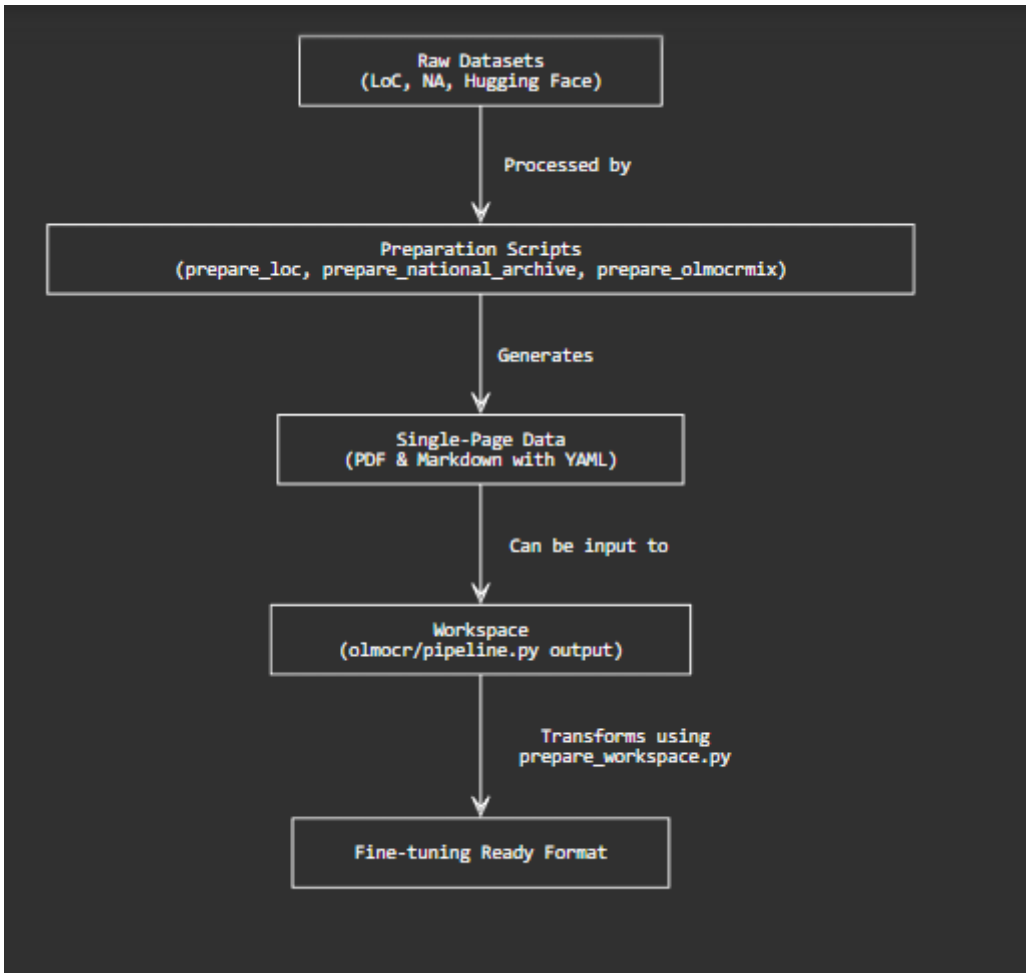
The framework provides utilities for preparing diverse external datasets. For instance, transcriptions from the Library of Congress and National Archives are processed by downloading images, converting them to PDF, and structuring transcriptions into Markdown and single-page PDFs. Similarly, **OLMoCR-mix** datasets from Hugging Face are downloaded, tarballs are extracted, and Parquet files are parsed to generate Markdown files with symbolic links to corresponding PDFs. These processes are largely handled by scripts within **olmocr/data**,
including **olmocr/data/prepare_loc_transcripts.py**, **olmocr/data/prepare_national_archive_transcripts.py**, and **olmocr/data/prepare_olmocrmix.py**.

Before using documents for training or evaluation, **olmOCR** employs advanced filtering mechanisms. PDF documents can be filtered based on criteria such as form detection, "download spam" content, language identification, and text validity checks. The system also assesses the textual coherency of documents using a causal language model. This ensures that only high-quality and relevant data are used. These filtering and coherency assessment capabilities are primarily implemented within the **olmocr/filter** directory, with the **PdfFilter** class in **olmocr/filter/filter.py** providing the core filtering logic and **olmocr/filter/coherency.py** handling textual coherency calculations via **get_document_coherency**.

The system also supports interaction with external services, notably the OpenAI Batch API. This involves generating batch requests from sampled PDF pages, orchestrating the submission of these requests, managing job lifecycles (uploading, creating, and downloading results), and processing the returned data into structured formats. This enables the generation of "silver data" by sampling PDF documents and pages, rendering them as base64 images, extracting anchor text, and building OpenAI chat completion prompts. The processed results are then analyzed to extract insights and summary statistics. Scripts like **olmocr/data/buildsilver.py**, **olmocr/data/runopenaibatch.py**, and **olmocr/data/process_openai_batch_results.py** in **scripts/data** and **olmocr/data** m anage these aspects.
Furthermore, **scripts/data/buildtestset.py** within **scripts/data** constructs test sets of single-page PDFs and their PNG renderings.

Finally, the framework includes processes for cleaning OCR transcriptions, often leveraging external Large Language Models (LLMs), which can incorporate visual context from PDF pages. The subsequent transformation of this processed data into fine-tuning-ready formats for various models, including handling metadata and prompt regeneration, is also supported. This functionality is found in **olmocr/data/clean_olmocrmix.py** within **olmocr/data**, and transformation scripts like **scripts/data/convertsilver_birr.py** and **scripts/data/convertsilver_openai.py** in **scripts/data**. For a deeper understanding of prompt construction and schema enforcement for LLMs, refer to LLM Prompt Construction and Schema Enforcement.

Diverse Dataset Preparation and Ingestion



The olmOCR project includes a suite of utilities located in the olmocr/data directory designed to prepare and ingest various external datasets, transforming them into a standardized format compatible with the olmOCR framework. This process typically involves downloading raw data, converting image files to single-page PDFs, extracting and structuring transcriptions into Markdown with YAML front matter, and managing metadata and symbolic links.

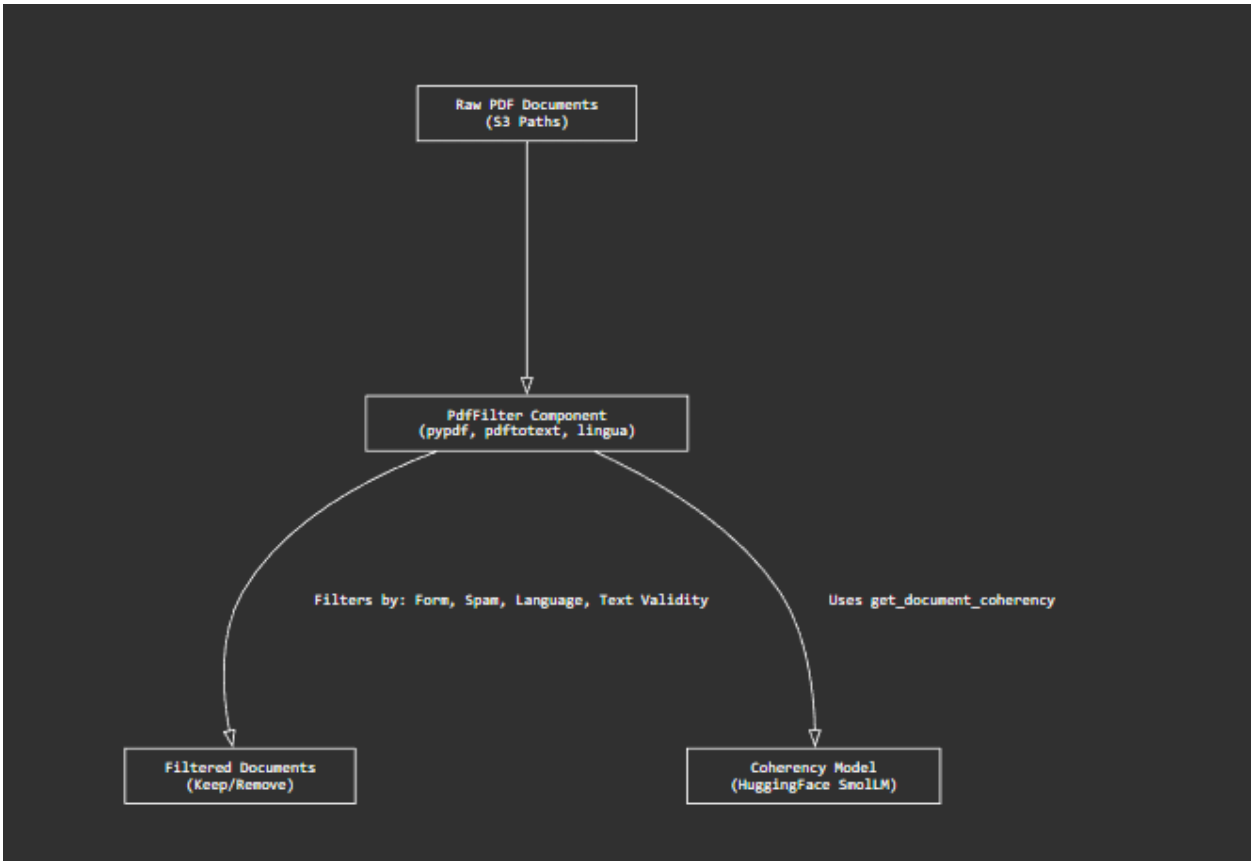
One of the primary functions is the preparation of the OLMoCR-mix dataset. The script in olmocr/data/prepare_olmocrmix.py downloads specified datasets from Hugging Face, intelligently filtering files based on parameters to optimize download size. It extracts PDF tarballs in parallel, parses Parquet files to derive document metadata, and then generates Markdown files with front matter containing this metadata and the extracted text. Crucially,

it creates symbolic links from the generated Markdown file's location to the corresponding extracted PDF, maintaining a structured dataset where each Markdown file is associated with its visual context. This modular approach allows for flexible handling of diverse OLMoCR-mix components and manages file organization with respect to dataset_path variations.

Similarly, historical document datasets, such as those from the Library of Congress and the National Archives, are processed for ingestion. The olmocr/data/prepare_loc_transcripts.py script downloads images from Library of Congress CSV records, converts them into single-page PDFs, and saves associated transcriptions as Markdown files. It incorporates robust features like retry logic for downloads and idempotency checks to avoid reprocessing existing files. For National Archives data, olmocr/data/prepare_national_archive_transcripts.py processes JSONL records, filters out access-restricted or AI-generated content, downloads relevant images (optimizing formats like TIFF/JP2 to JPEG for size), and then converts them to PDFs, pairing them with extracted transcriptions in the olmOCR Markdown format. These scripts ensure that disparate historical document formats are standardized, making them accessible for olmOCR's evaluation and training pipelines.

The resulting structured data, often from these ingestion processes, can then be further prepared for model fine-tuning. The olmocr/data/prepare_workspace.py utility processes JSONL output files, which encapsulate processed document information. For each entry, it extracts specific pages from source PDFs, saves them as individual single-page PDFs, and generates corresponding Markdown files. These Markdown files include rich YAML front matter populated with page-level attributes such as language, rotation information, and content classification (e.g., table or diagram), along with the extracted text. This fine-grained page-level output ensures that models can be trained and evaluated on precise document segments with comprehensive metadata. The workflow supports both local and S3 storage for source PDFs, leveraging caching to improve efficiency.

PDF Filtering and Textual Coherency Assessment

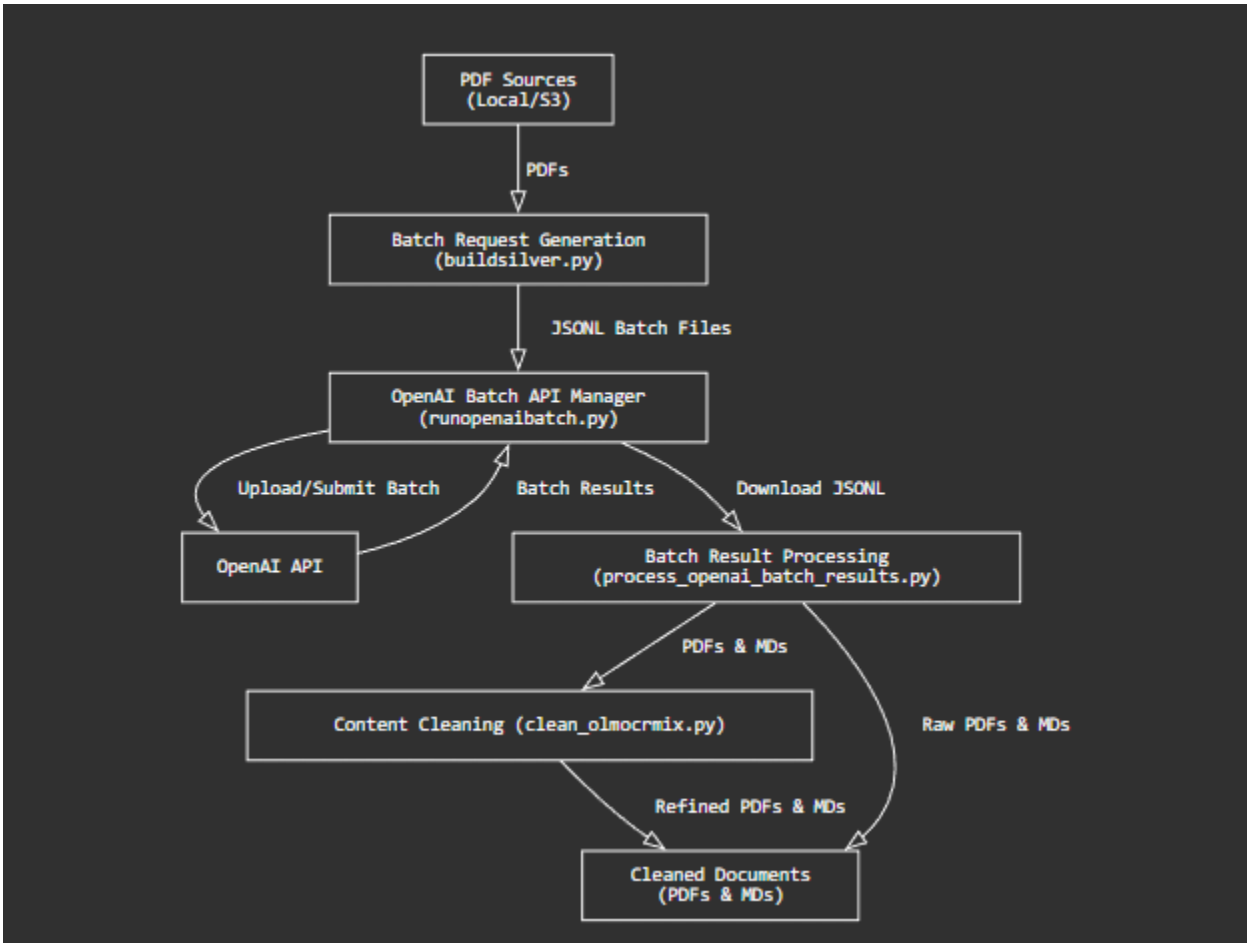


The olmocr project includes capabilities for filtering PDF documents based on various criteria and for assessing their textual coherency. This ensures that only relevant and high-quality documents are used in downstream processing.

The PdfFilter class, found in olmocr/filter/filter.py, manages the filtering process. This class can detect if a PDF contains interactive forms, which might indicate that it is not a standard document for OCR. It also implements "download spam" detection by analyzing the frequency of specific keywords within the document's text, helping to identify and discard unwanted promotional content. Language identification is another key feature, allowing the system to retain only documents written in specified languages. Basic text validity checks, such as minimum text length and the proportion of alphabetic characters, are also performed to filter out PDFs with poor or unreadable text extraction.
The PdfFilter leverages external tools like pdftotext for robust text extraction and the lingua library for language detection.

Beyond filtering, olmocr can assess the textual coherency of documents.
The get_document_coherency function, located in olmocr/filter/coherency.py, uses a pre-trained causal language model (e.g., HuggingFaceTB/SmolLM-135M) to calculate the average log-likelihood per token for a given text. This metric provides an objective measure of how well the language model predicts the text, with higher scores generally indicating greater textual coherency. For documents longer than the model's context window, the text is chunked and processed incrementally. This coherency assessment helps in maintaining data quality by identifying documents that may have fragmented or nonsensical text.

OpenAI Batch API Integration and Management



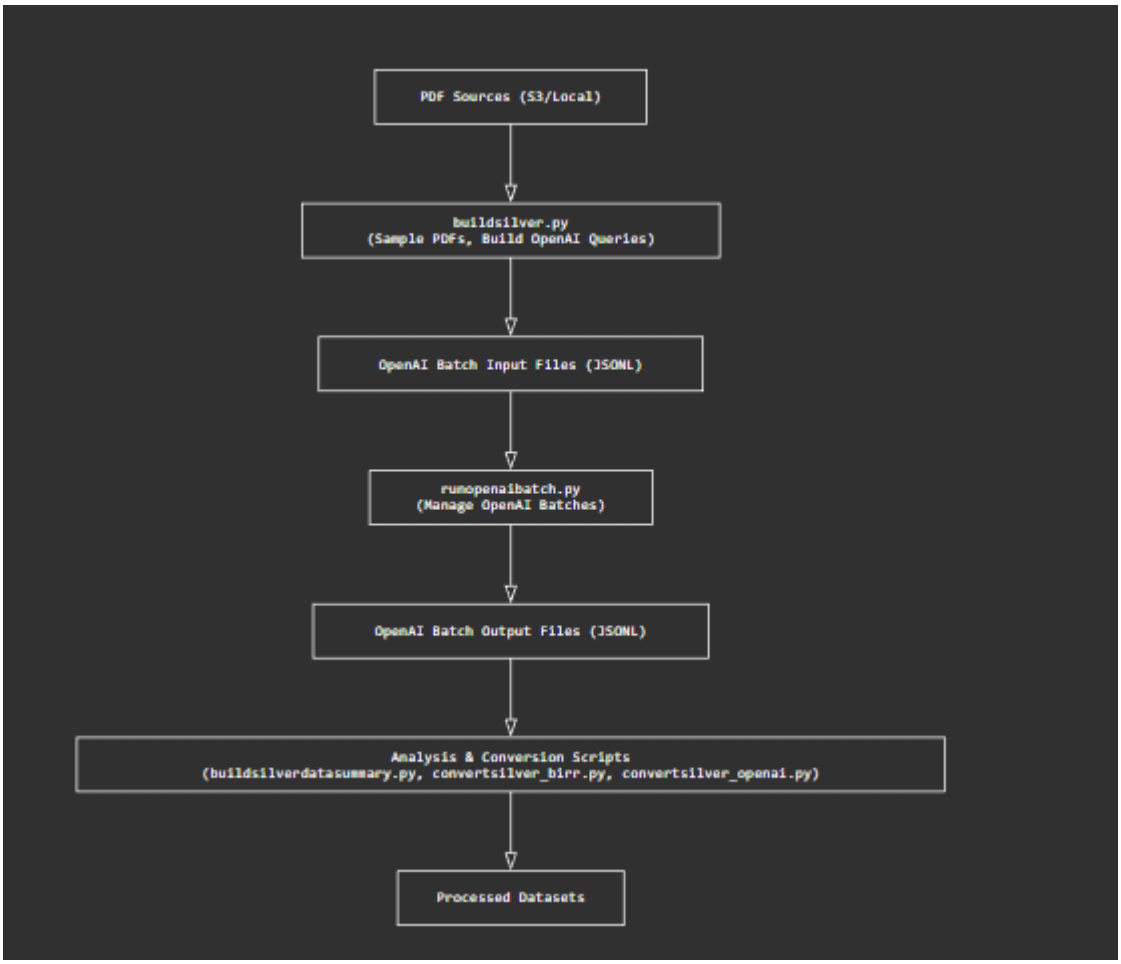
Integration with the OpenAI Batch API involves a comprehensive workflow, from generating initial requests to processing final results. This process begins with generating batch API requests from sampled PDF pages. The olmocr/data/buildsilver.py and

scripts/data/buildsilver.py scripts are used to create "silver data" by processing PDF documents from various sources, including local file systems and S3 buckets. These scripts sample pages, render them as base64 encoded PNG images, extract anchor text, and format these components into structured JSON objects suitable for OpenAI chat completions. This includes specifying structured outputs via openai_response_format_schema and including logprobs for detailed analysis of model responses.

Once batch requests are generated, the system orchestrates their submission to the OpenAI Batch API. The scripts/data/runopenaibatch.py and olmocr/data/runopenaibatch.py scripts manage the entire lifecycle of these batch jobs. This includes uploading .jsonl files to OpenAI, initiating batch jobs, and monitoring their progress. A critical aspect of this management is handling OpenAI's disk space limitations, which the system addresses by maintaining a persistent processing state for each file using a local JSON file named UPLOAD_STATE_FILENAME. This state management enables the system to resume operations, track progress, and efficiently manage disk space by pausing new uploads if a predefined maximum (defaulting to 25GB) is approached. It also includes utilities for downloading completed results and, optionally, for deleting files stored in the user's OpenAI account.

After the batch jobs are completed and results are available, the final step involves processing the returned data into structured formats. The olmocr/data/process_openai_batch_results.py script automates this post-processing. It reads the raw JSONL output from OpenAI, parses each response, and then organizes these results. For each successful response, the script extracts key information such as primary language, rotation validity, and natural text. This extracted content is then formatted into a FrontMatter-style Markdown file, and the original PDF associated with the response is copied to an organized output directory, mirroring the original directory structure. This ensures that the processed data is both human-readable and machine-parseable, ready for further analysis or fine-tuning of models. Additionally, the olmocr/data/clean_olmocrmix.py script can clean OCR transcriptions using ChatGPT, leveraging visual context from PDF pages. It processes olmocr-mix formatted data, sending the OCR text and PDF image to ChatGPT for correction, and stores the enhanced output in a structured format. This integration with AI models enhances the quality of the OCR transcriptions.

Silver Data Generation and Analysis



The creation and analysis of "silver data" within the **olmOCR** project involves sampling PDF documents and pages, rendering them as base64 images, extracting anchor text, and constructing OpenAI chat completion prompts. This process is primarily handled by the script **scripts/data/buildsilver.py**, which is designed to generate **openai_batch_data** for downstream processing. The system gathers PDF paths from local files or S3, applies filtering to exclude unwanted documents, and then samples specific pages from each PDF. For each selected page, it generates a base64 encoded PNG image and extracts contextual "anchor text." This information is then used to build structured OpenAI chat completion requests, formatted for optimal use with the OpenAI Batch API and designed to produce structured outputs. These generated requests are written to JSONL files, managed to adhere to file size limits for batch processing, and processed in parallel to enhance efficiency.

Following the generation of this data, the script **scripts/data/buildsilverdatasummary.py** analyzes the output to extract insights and summary statistics. This analysis includes parsing **custom_id** values from the generated JSONL files to identify PDF hashes. These hashes are then mapped to Uniform Resource Identifiers (URIs) using a pre-cached Athena CSV, which is stored in an SQLite database for efficient lookups. The script extracts domains from these URIs and compiles summary statistics on domain occurrences, generating reports such as the top 1000 domains and a sample of the processed data. This analysis step helps in understanding the characteristics and distribution of the generated silver dataset.

The silver data can also undergo further transformations for fine-tuning purposes. For example, scripts like **scripts/data/convertsilver_birr.py** and **scripts/data/convertsilver_openai.py** process these JSONL files, renaming and extracting specific fields. These scripts can optionally rewrite fine-tuning prompts by re-extracting information from the original PDFs, regenerating the prompt text, and updating image URLs. This allows for the creation of fine-tuning-ready formats for various models, including handling metadata and prompt regeneration. The entire process of submitting and managing these generated JSONL files with the OpenAI Batch API, including uploads, job creation, result downloads, and state persistence, is orchestrated by **scripts/data/runopenaibatch.py**. See OpenAI Batch API Integration and Management for more details.

OCR Transcription Cleaning and Data Transformation

The process of cleaning OCR transcriptions within the **olmocr** project often involves leveraging large language models (LLMs) like ChatGPT, which are provided with visual context from PDF pages to enhance accuracy. This approach addresses the common inaccuracies inherent in raw OCR output by allowing an LLM to "see" the document's layout, tables, diagrams, and formatting, rather than relying solely on the extracted text.

The workflow begins by identifying pairs of Markdown files containing initial OCR transcriptions and their corresponding single-page PDF counterparts. For each pair, the PDF page is rendered into a base64 encoded PNG image. This image, along with the original Markdown content, is then sent to ChatGPT. A detailed system prompt guides ChatGPT to perform specific cleaning tasks and return structured output. This structured output is enforced using Pydantic models, such as **CleanedDocument** defined in **olmocr/data/clean_olmocrmix.py**, which specifies fields like **cleaned_text**, **confidence_score**, **corrections_made**, and classifications such as **is_table** or **is_diagram**. This ensures that the LLM's response is consistent and easily parseable.

Once cleaned by ChatGPT, the processed data is transformed into formats suitable for fine-tuning various machine learning models. This involves generating new Markdown files for the cleaned text, complete with YAML front matter that includes metadata extracted from the LLM's response (e.g., **primary_language**, **is_rotation_valid**). Symbolic links are also created to maintain traceability to the original PDF and Markdown files.

Further data transformation for fine-tuning involves processes detailed in **scripts/data/convertsilver_birr.py** and **scripts/data/convertsilver_openai.py**. These scripts take the JSONL outputs, which might contain **OpenAI** batch API results or other intermediate data, and restructure them into a format ready for model consumption. This transformation can include regenerating prompts by re-downloading PDFs from S3, rendering pages, extracting anchor text, and rebuilding the prompt content using functions like **build_finetuning_prompt**. This ensures that the fine-tuning data is consistent and incorporates the visual context necessary for robust model training. The scripts also manage metadata and can generate prompt length histograms for analysis. For further details on the preparation of various external datasets, see Diverse Dataset Preparation and Ingestion.