ChatGPT

Additional Considerations and Sections

Official PMC Data Download via NCBI Developer Tools

For comprehensive and legally compliant data retrieval, we will leverage **official PMC tools** rather than ad-hoc scraping. The NCBI **PMC Developer Resources** include services like the PMC **FTP Service**, Cloud AWS buckets, E-utilities, and OAI-PMH, which are the *only* approved methods for bulk downloading PMC content [1]. In particular, the **PMC Open Access subset** can be obtained in bulk XML or plain text via the FTP service's baseline packages (released ~twice a year) [2]. We plan to use these services (e.g. downloading the Open Access baseline) to gather case report articles at scale. For faster access, NCBI also offers an **AWS S3** open data bucket for the PMC OA subset without login requirements [3]. Using these official channels ensures we adhere to PMC's policies and have up-to-date content. At a small scale (for initial prototyping), we might start with targeted E-utility queries or small subsets; but ultimately **full-scale processing** will involve downloading the **entire PMC OA baseline** dataset for the relevant period (e.g. the mid-June or mid-December release) for maximum coverage. All downloads will include metadata like PMCID/PMID/DOI mapping (via files like `PMC-ids.csv.gz` on the FTP) to help link articles with our records [4].

Using OpenRouter Free LLM Models Instead of Local Ollama

Instead of running local Ollama LLM models, we will integrate **OpenRouter.ai** to access **free cloud-based models**. OpenRouter provides a unified API endpoint to many different large language models (LLMs) – acting like a "router" that gives one consistent interface to models from OpenAI (GPT-3.5, GPT-4), Anthropic (Claude), Google (Gemini), Meta (LLaMA family), and others [5]. We specifically plan to utilize **free models** available through OpenRouter (filtering for `max_price=0`), meaning we can call high-quality open-source models at no cost. OpenRouter's platform offers an *impressive selection of free, high-quality models that deliver powerful capabilities without cost barriers* [6]. By using OpenRouter, we avoid the complexity of hosting large models locally and benefit from their managed infrastructure (no need for a powerful local GPU). The user will only need to obtain an OpenRouter API key (sign-up is free) and our code can then route requests to the chosen free model. This approach also grants flexibility: we can experiment with different LLMs (e.g. various sizes of LLaMA-2 or other community models) simply by changing the model name in the API call. In summary, **OpenRouter** replaces local Ollama usage, providing convenient access to multiple LLMs via cloud – *with zero-cost models* – and simplifying deployment.

PDF Download and Manual PDF Import Options

Not all articles may have their full text readily accessible via API or XML, so our pipeline will handle PDFs with care. The user will be **prompted whether to download PDFs automatically** for each reference or not. If a paper is in the PMC Open Access subset, we might retrieve its XML or even PDF (PMC provides individual PDFs for some open-access articles, but only those under certain licenses [7]). For cases where the PDF is not available through PMC (e.g. non-OA articles or those behind paywalls), the user should have the option to **manually provide the PDF** instead. This means our interface or script will ask, *"Do you want to attempt downloading the PDF for this article?"* – if the user opts *Yes*, the system will try an automated fetch (using

known open repositories or direct links when possible); if *No*, the process can pause for the user to upload or place the PDF in a specified directory. This design gives the user control, ensuring we respect any access restrictions and avoid unwanted automated downloads. All fetched or user-provided PDFs will be stored in a dedicated folder (e.g. `data/pdfs/`), and we will **name each PDF by its PMID or DOI** for clear linkage. Using a consistent naming convention (like `PMID_32679198.pdf`) will allow the code to automatically match PDFs to the corresponding article records via identifiers. In summary, the system will support both **automated PDF retrieval (with user consent)** and **manual PDF import**, to flexibly accommodate different access scenarios while keeping the process transparent to the user.

Processing of CSV Files (Clinician Input, Clinician Output, Lens.org Export)

We have multiple CSV files involved, serving different roles, so clear instructions are needed on how to use each:

- **Clinician Input CSV** – This is the table that was originally given to the clinician. In this file, each row corresponds to a **paper (case report article)**, and it likely contained preliminary info or placeholders to be filled (e.g. article title, maybe some known fields, and blank columns for the clinician to fill in details). This input CSV can be used to identify the list of target papers. However, since one paper can describe multiple patients, this CSV is organized per article (not per patient). It may contain an Article ID (PMID or DOI) which we can use to retrieve the article's text and to link with the output.

- **Clinician Output CSV (Manually Processed)** – This CSV is the **ground truth dataset** created by the clinician after reading the papers. Here, **each row corresponds to a single patient** case. In other words, if one journal article contained 3 patient case reports, that article would appear as 3 separate rows in this output (one for each patient). The output CSV has columns with all the extracted information (demographics, symptoms, diagnoses, outcomes, etc.) for each patient – all of these columns can be considered *ground truth* values that our automated pipeline should ideally reproduce [8] [9]. It also typically includes a reference identifier (e.g. a PMID or DOI column) to indicate which article the patient came from, and a patient identifier (like "Patient 1", "Patient 2", etc). **We will treat this output CSV as the gold-standard reference** for evaluating our extraction. Processing-wise, we will likely load this CSV into a dictionary keyed by patient ID (concatenating article ID and patient number if needed) to compare with the model's output [10] [11]. All fields in this file are final curated data, so no further cleaning is needed beyond ensuring consistent formatting.

- **Lens.org Export CSV** – This file contains metadata about the articles (exported from Lens.org as an example). It serves **only as a reference for metadata organization** and to highlight what metadata is important. The lens.org export might include fields such as article title, authors, journal, publication year, DOI, PMID, open access status, etc. We do not strictly need to input this file into the pipeline code; instead, we use it as a guide. For example, by reviewing the Lens export, we can ensure our system captures key identifiers (PMID, DOI) and perhaps confirm that we have all target papers accounted for. If needed, we might also use it to fill in any metadata gaps (for instance, if the clinician input lacked DOIs, we could map PMIDs to DOIs using the lens data). However, **the primary pipeline will function without requiring the lens CSV**, as its role is informational. It has shown us how well-organized metadata can look, which we aim to emulate in our outputs (e.g. each patient

entry in final output should carry the PMID/DOI to link back to source, plus any other relevant metadata like perhaps the paper title or year for context).

Linking by PMID/DOI: A crucial step in processing these files is to **match patients to their source articles via a stable identifier**. We will use PMID or DOI as the linking key (both are present in the CSVs). For example, when reading the ground truth CSV, we collect each patient row under its PMID. Then, when processing the article text for that PMID, we know which ground truth records to compare against. If we download PDFs or XMLs, we will name and organize them by PMID/DOI as well. This automatic linking is facilitated by the consistency of identifiers – indeed, PMC provides a mapping file for PMCID-PMID-DOI which underscores the importance of using these IDs for integration [4]. All data (input, output, and any fetched content) should be placed in a **dedicated directory structure** (see file structure below) to keep things organized and allow programmatic matching. By following these procedures, we ensure that each patient entry in the output CSV is correctly associated with the corresponding article text during extraction and evaluation.

Required Files and Data Resources to Download in Advance

To set up the project, we should gather several **files and datasets beforehand**. Below is a checklist of all necessary resources and how to obtain them:

- **PMC Open Access Subset (Baseline)** – As decided, we will use the PMC Open Access articles for text mining. You should download the **PMC OA baseline package** (in XML format) from the official PMC FTP or AWS service [2]. This baseline contains hundreds of thousands of articles in machine-readable form. If focusing only on specific case reports (e.g. pertaining to a certain disease), you may alternatively download a filtered set or individual articles, but ultimately having the full baseline will allow scalability. *(Note: The baseline is updated biannually; choose the latest release. After download, unzip the package to get .xml files.)*

- **NCBI Mapping File (PMC-ids.csv)** – Download the `PMC-ids.csv.gz` from the PMC FTP root [4]. This file maps PMCID, PMID, DOI, and other IDs for all PMC articles. It will be useful for quickly converting between IDs or confirming that we have the correct identifiers for each article.

- **Clinician Input/Output CSVs** – Have the CSV provided to the clinician (input) and the manually curated output CSV ready in the `data/input/` folder (or designated input directory). These are typically provided by the user (you have them from the project assets). No external download is needed beyond ensuring you have the latest version from the clinician. Place them in our project structure for easy access.

- **Lens.org Metadata Export** (if applicable) – If a Lens.org export CSV was generated (e.g. a list of all case report articles with metadata), have this file available as well (in `data/input/` or perhaps a `data/ref/` folder since it's reference metadata). Again, this likely comes from the user's prior work – if not already on hand, one can reproduce it by using Lens.org's export feature for the list of relevant publications.

- **Ontologies and Controlled Vocabularies**: We will incorporate medical ontologies to normalize and validate extracted information.

- **HPO (Human Phenotype Ontology)**: Download the latest HPO release, preferably in OBO or JSON format (e.g. `hp.obo` from the official HPO site). The user has indicated they will provide this file to the coding agent, so ensure it is placed in the project (e.g. in `data/ontologies/hp.obo`).
- **MONDO (Monarch Disease Ontology)**: Download the MONDO ontology OBO file (e.g. `mondo.obo` from OLS or OBO foundry). This will be provided by the user as well and should reside in `data/ontologies/mondo.obo`. MONDO can help standardize disease names (for instance, mapping "Leigh syndrome" to a MONDO ID).
- **SNOMED CT**: This is a comprehensive clinical terminology we might use for encoding diagnoses or findings. **SNOMED CT is not freely downloadable without a license.** To obtain SNOMED, you need to have a UMLS license or be in a member country. The recommended approach is to create a UMLS account and download the SNOMED CT International edition release from the NIH/NLM website [12] (or from the SNOMED International portal). Because of the licensing, the user must either download SNOMED manually (and agree to terms) or use an API provided by UMLS. We can include instructions or a script to load SNOMED if the file is available (for example, using **PyMedTermino** or the SNOMED CT **Snowstorm API** if internet access is possible). *If SNOMED CT cannot be obtained, the pipeline should still run (perhaps skipping SNOMED-specific normalization), but for best results, having it is desirable.* Place SNOMED data files (perhaps the RF2 release files or a processed subset) in `data/ontologies/snomed/`.

- **Other references**: If there are any gene databases or additional vocabularies needed (the output CSV has a "gene" column, so we might need a gene list or just rely on text), consider if any resource like OMIM or HGNC is required. At this stage, we have not explicitly listed one; presumably gene extraction will be handled via text/LLM and validated via known gene names. If needed, we could download a list of known gene symbols (from Ensembl or HUGO) to cross-check gene outputs, but this is optional.

- **Software/Utilities**: While not "files", ensure any required external tools are installed. For example, if using a PDF-to-text tool like Poppler's `pdftotext` (as in the current pipeline code), the binary should be installed on the system. Alternatively, if we use a Python PDF library (like PyMuPDF), that will be covered by `pip` requirements. Also, if using NCBI E-utilities heavily, it's wise to have an NCBI API Key (you can obtain one from NCBI by creating an account) to increase request limits – this key would be set in an environment variable or in the code when calling NCBI APIs.

Before running the pipeline, verify that all the above files are in place. In summary, the **files to download/provide beforehand** include the *PMC OA dataset*, ID mapping CSV, the *clinician CSVs*, the *Lens metadata CSV*, the *HPO OBO*, *MONDO OBO*, and *SNOMED CT files* (if available). Having these ready in the appropriate folders will ensure the pipeline can find everything it needs without manual interruption. (The next sections outline environment setup and project structure for where these files should reside.)

Virtual Environment Setup, Requirements, and Environment Variables

Setting up a clean Python environment will help manage dependencies. We will use a **virtual environment (venv)** and prepare a `requirements.txt` listing all needed Python packages. Below are the steps and an overview of the requirements:

1. **Python Version**: Ensure you have Python 3.x (preferably 3.9+ for compatibility). Create a virtual environment: for example, `python3 -m venv venv` and activate it (`source venv/bin/activate` on macOS/Linux, or using the appropriate command on Windows).

2. **Requirements**: Install the necessary packages via pip. We will provide a `requirements.txt` file. Key libraries likely include:

3. **pandas** – for CSV reading/writing and data manipulation.
4. **requests** – for making HTTP calls (to download files or call APIs). This may be used for fetching PMC data or hitting the OpenRouter API if not using a specialized SDK.
5. **openai** – OpenRouter is compatible with the OpenAI API interface [13], so we can use OpenAI's Python SDK to send requests. This saves us from writing custom request code for each model. The OpenAI package will allow us to specify the API base URL as OpenRouter and use our key.
6. **python-dotenv** – to easily load environment variables (like API keys) from a `.env` file.
7. **PyMuPDF (fitz)** or **pdfminer.six** – a library for PDF text extraction, if we need to parse PDF files directly. (Alternatively, if we stick to using external `pdftotext`, we may not need a Python PDF library, but including one is useful for cross-platform consistency. PyMuPDF is a good choice for reliability and speed.)
8. **lxml** – for parsing XML files (if we use the PMC OA XMLs, lxml can help navigate the article structure to extract sections of text).
9. **regex** (built-in `re` or possibly the `regex` module) – since part of our extraction logic may rely on regular expressions to find age, gender, etc. (The Python standard `re` is sufficient, so an extra module may not be needed).
10. **numpy** (possibly) – if needed for numeric operations or array handling.
11. **tqdm** – if we want progress bars for processing many files.
12. **Ontologia or OWL libraries** (optional) – If we plan to directly query OBO files, we might use a library like `pronto` or `owlready2`. This is optional; we could also parse the ontologies with simple code or use them in memory as needed. If the plan is just to load HPO/MONDO terms into a set or dict, pandas or basic file I/O might suffice.
13. **PyMedTermino2** (optional for SNOMED) – a library that can interface with SNOMED CT if the data is available. This could simplify finding SNOMED concepts. However, it requires the SNOMED release files and can be complex. We might instead opt for a simpler approach (like searching within provided SNOMED descriptions files using Python) so including this is not mandatory.

We will pin versions of these packages as appropriate in `requirements.txt`. After creating the venv, the user will run `pip install -r requirements.txt` to fetch all dependencies.

1. **Environment Variables**: We will use a `.env` file (and possibly an `env.example` template) to manage API keys and configurable settings. For instance, create a file named `.env` in the project root with content:

   ```
   OPENROUTER_API_KEY=your-api-key-here
   ```

   This key is obtained from your OpenRouter account (once signed up). Our code will load this environment variable to authenticate requests. If using the OpenAI SDK for OpenRouter, we also need to set the base URL. This can be done in code (e.g., `openai.api_base = "https://openrouter.ai/api/v1"`), or potentially via environment (OpenAI's library sometimes reads `OPENAI_API_BASE` if set). Other environment configs might include:
2. `NCBI_API_KEY` – your NCBI Entrez API key, if you have one, to use with E-utilities.
3. `OPENROUTER_API_BASE` (if needed, or we handle in code as mentioned).
4. Any file paths or modes (though generally we'll keep paths in a config section of the code rather than env).
5. Flags like `DOWNLOAD_PDFS=yes/no` could also be environment-driven, but since we plan to prompt the user, that might not be necessary as an env variable.

We will provide an `env.example` file that lists these variables without values (for the user to fill in their actual keys). This ensures that sensitive keys are not hard-coded in the code or shared via version control. The user should rename it to `.env` and input their keys before running.

1. **Activating and Running**: Once the environment is set and packages installed, the user can run the pipeline (e.g., via a main script or Jupyter notebook). The environment setup ensures all required libraries are available and keys loaded.

In summary, **create a virtual environment**, **install the listed requirements**, and **configure the `.env` file** with your API keys. This will prepare the groundwork to execute the project's code. (All these instructions will be included in a README as well for clarity.)

Project Architecture Overview (Pipeline Design)

We will implement a modular pipeline to extract and process information from case report articles. The architecture draws inspiration from recent research on LLM-based extraction, which advocates breaking the task into controllable steps [14]. Here is an overview of the main components of our project's architecture:

1. **Data Acquisition and Ingestion:** In this initial phase, the system gathers the raw textual data of case reports.
2. We start with the list of target articles (from the clinician's input CSV or other source). For each article, if we have the PMC XML (from the baseline download), we will load it; otherwise, if only a PDF is available, we convert the PDF to text.

3. The ingestion step thus involves either parsing XML (to extract the relevant sections of the paper, e.g. case descriptions) or running a PDF-to-text extraction. The output is the full text (or segmented text) of each article in a machine-readable form.
4. As part of ingestion, we may also **segment the text by individual patient** if the article contains multiple cases. For example, many case reports enumerate patients (Patient 1, Patient 2, etc.). We can use simple heuristics or section titles to split the text accordingly (our earlier prototype `segment_patients` function does this using clues like "Patient 1" headings). This yields each patient's narrative separately.

5. We ensure that each text segment is tagged with the PMID/DOI and a patient identifier (e.g. "PMID12345_Patient1") so it can be traced through the pipeline.

6. **LLM-based Information Extraction:** Next, we use a **Large Language Model** (via OpenRouter) to extract structured information from each patient's text.

7. We will design prompt templates that instruct the LLM to read a patient case description and output specific fields (e.g. sex, age of onset, clinical symptoms, genetic mutation, outcome, etc.). The prompt might look like a request: "Extract the following fields for the patient described: Age of onset, Sex, Genetic cause, … If not mentioned, leave blank."
8. Depending on the model's capabilities and context length, we might process one patient at a time. The LLM's output will be parsed (e.g., if we format the answer as JSON or a list).
9. We may incorporate a **few-shot chain-of-thought approach** as seen in literature [15] – for instance, giving an example of a filled template to guide the model, and using a structured output format to minimize hallucination.
10. This step might be broken into sub-steps (similar to the Wang *et al.* pipeline [14]): first ask the LLM to identify key concepts (phenotypes, diagnoses) in the text, then ask specific questions. However, given our scale might be smaller, an end-to-end extraction in one prompt per patient could suffice. We will iterate on prompt design as needed to ensure the model outputs all required fields with accuracy.

11. The output from this LLM extraction is a structured **patient record** (in memory, likely as a Python dict or DataFrame row). Each record includes the article ID and patient ID along with the extracted attributes.

12. **Ontology Mapping and Normalization:** After raw extraction, we will apply ontology resources (HPO, MONDO, SNOMED) to normalize and validate the data:

13. **Phenotype terms (HPO):** If the LLM outputs symptoms or phenotypic descriptions (e.g. "hypotonia, developmental delay"), we will map these to HPO IDs. This could be done by string matching to HPO term names or using an algorithm to find the closest HPO term. We have the HPO OBO file, so we can create a dictionary of term names to IDs for lookup. The output might then include standardized HPO codes for each phenotype mentioned.
14. **Disease diagnosis (MONDO/SNOMED):** For the final diagnosis or disorder (e.g. "Leigh syndrome"), we will use MONDO to retrieve a normalized identifier. Similarly, SNOMED CT could be used for any clinical findings or diagnoses. *For example, Wang* et al. *used SNOMED CT to format extracted concepts and filtered out any terms that weren't in the SNOMED vocabulary* [16]. In our case, if the model gives a diagnosis name, we check it against MONDO ontology to get an ID, ensuring consistency (and catching any spelling variants).

15. **Gene names:** If a gene is extracted (e.g. “MT-ATP6”), we might cross-verify it against a known gene list or database to ensure correctness. This could be as simple as checking it matches the official gene symbol format or looking it up via an API (like MyGene.info) if internet is available. Given our domain (Leigh syndrome is often caused by mitochondrial DNA mutations), we expect gene names or variants in the text which we can standardize (e.g. ensure “MT-ATP6” is consistently formatted).
16. **Mortality/Outcome:** The binary outcome (“0=alive, 1=dead” in the CSV) will be extracted by rules or LLM. We just ensure it’s captured correctly as 0 or 1. No ontology needed, but maybe ensure consistency (like if LLM outputs “alive” vs “deceased”, we convert to 0/1).

17. Essentially, this stage acts as a **post-processing filter**: we take the LLM’s output and run checks or mappings to correct any non-standard terminology. By grounding terms in known ontologies, we increase the reliability of the data and reduce model hallucinations [17].

18. **Aggregation and Output Generation:** Once each patient’s record is extracted and normalized, we aggregate the results.

19. We will compile all patient records into a structured format (likely a CSV or JSON). The fields will align with the schema of the ground truth CSV (our code can load a schema definition, e.g. from `table_schema.json`, to know expected field names and types).
20. The output can be a CSV file (for easy comparison with the clinician output) and/or a JSON for further use. We will ensure that each record in the output contains at least: PMID, Patient ID (or some unique key), and all the information fields (age of onset, sex, etc., including normalized ontology IDs where appropriate).
21. If multiple articles were processed, the output file will contain multiple patients grouped by article. We might sort or group the output by PMID for clarity.
22. Additionally, this is where we can compare with ground truth (if doing evaluation). We’ll have a routine to read the clinician output CSV and align each patient record by patient ID and article, then compute field-wise accuracy or differences [11] [18]. Any discrepancies can be logged for analysis (as was done in the prototype code).

23. Finally, the results could be fed back to the user (for instance, showing a summary: “Extracted X patients from Y articles. Accuracy: …” or highlighting which fields were missed).

24. **Iterative Refinement:** The architecture allows for iterative improvement. For example, if the LLM makes consistent errors in a certain field, we can adjust the prompt or add a post-processing rule. The modular nature (ingestion → LLM extraction → normalization → evaluation) makes it easy to identify which stage to tweak. This also aligns with a *retrieval-based CDS* approach where relevant info is first identified, then queried, etc., though our scale is smaller than something like the PMC-Patients benchmark.

Overall, the project architecture is a pipeline that flows from **data input to structured output**, augmented by LLM intelligence and domain knowledge bases. The use of **multiple controlled steps** (instead of one giant black-box prompt) is deliberate to maintain control and accuracy [14]. By the end, we expect to automatically produce a table of patient case information that closely matches the clinician’s manual output, demonstrating the effectiveness of the approach.

Proposed Project Folder Structure

Organizing the project files systematically will make the workflow reproducible and maintainable. We suggest the following folder structure (inspired by best practices for data science projects [19], separating inputs, code, outputs, etc.):

```
project_root/
├── input/                # Raw input data and reference materials (read-only)
│   ├── clinician_input.csv         # The table initially given to clinician (by
paper)
│   ├── clinician_output.csv        # Manually curated output from clinician (by
patient)
│   ├── lens_export.csv            # (Optional) Metadata export from lens.org
│   └── articles_list.txt          # (Optional) List of article IDs to process
(if not using CSV)
├── data/                 # Data storage for processing (writable)
│   ├── pdfs/                      # PDFs of articles (downloaded or provided)
│   ├── xml/                       # XML files of articles (if using PMC OA XMLs)
│   ├── ontologies/                # Ontology files (HPO, MONDO, SNOMED)
│   │    ├── hp.obo
│   │    ├── mondo.obo
│   │    └── snomed/ (folder for SNOMED release files if applicable)
│   └── intermediate/              # Intermediate processed data (if any, e.g.
extracted text, temp JSON)
├── src/                  # Source code for the project
│   ├── data_ingestion.py          # Code to download/parse articles (XML/PDF to
text)
│   ├── extraction.py              # Code for LLM prompting and extraction logic
│   ├── ontology_mapping.py        # Code for applying ontologies to normalize
terms
│   ├── evaluation.py              # Code to compare output with ground truth
│   ├── utils/                     # (optional) Utilities (e.g., helper
functions, constants)
│   └── __init__.py                # (if structuring as a package)
├── notebooks/            # Jupyter notebooks for exploration or prototyping (if
any)
│   └── EDA_and_tests.ipynb        # Example notebook analyzing a sample article,
etc.
├── output/               # Final output files and results
│   ├── extracted_patients.csv     # The CSV output generated by the pipeline
│   ├── extracted_patients.json    # (Optional) JSON version of the output
│   └── logs/                      # Any log files or discrepancy reports from
evaluation
├── env/ or config/       # Configuration and environment files
│   ├── requirements.txt           # Python dependencies
```

```
│   ├── requirements_dev.txt      # (Optional) dev dependencies (linters, etc.)
│   ├── .env.example              # Template for environment variables (API
keys, etc.)
│   └── config.yaml               # (Optional) configuration file for runtime
parameters
└── README.md             # Documentation and usage instructions
```

A few points about this structure: - The `input/` directory holds the original input datasets (CSVs from the clinician, etc.) and should be treated as read-only – we don't modify these files; they are source data. - The `data/` directory is for any data we fetch or generate during processing. We separate `pdfs/` and `xml/` for clarity: if we have XMLs from PMC, they go into `data/xml`, whereas any PDFs we had to download or were provided go into `data/pdfs`. Ontology files live in `data/ontologies` for easy access by the code. The `intermediate/` subfolder can store any temporary outputs (for example, if we split one article's text by patient and want to save those snippets, or if we output the LLM raw responses before normalization). - The `src/` directory contains all our Python modules. We break down functionality by file: e.g., `data_ingestion.py` for fetching/downloading articles and converting to text, `extraction.py` for running the LLM extraction (this might include prompt templates, the calls to OpenRouter, etc.), `ontology_mapping.py` for functions that load ontologies and map terms, and `evaluation.py` to compare results. This modularization makes each piece easier to manage and test. We can also include a main script (or use the `__main__` section) to orchestrate the pipeline (or even better, a CLI entry point). - Optionally, a `notebooks/` folder is included if we used Jupyter notebooks for analysis or debugging (for example, an exploratory analysis on one sample article, or visualizing something). This isn't required for running the pipeline but is good for documentation and reproducibility of any analysis. - The `output/` directory will hold the final results. The most important file will be `extracted_patients.csv` which is the machine-generated version of the clinician's table. We put it separate from input to avoid confusion. We might also output a JSON for easier programmatic use, or any logs (like a list of discrepancies between extraction and ground truth, or runtime logs). - The `env/` **or** `config/` area holds config files. We include the `requirements.txt` for setting up the environment, and a `.env.example` as discussed for environment variables. A `config.yaml` could be used to store configuration like which model to use on OpenRouter, what fields to extract, etc., making it easy to change without modifying code. This is optional but can be helpful if the user wants to tweak settings.

This structure ensures a clear separation between code, data, and results, which improves clarity and reproducibility [20]. For instance, raw data isn't mixed with generated output, and source code isn't mixed with analysis results. New collaborators (or the future you) can quickly navigate: they'll find data in `data/` or `input/`, code in `src/`, and final outputs in `output/`.

Note: Ensure to update the paths in the code to match this structure. For example, if the code expects ontology files in `data/ontologies/`, the file paths should be constructed accordingly (perhaps with a config pointing to that directory). Also, consider using relative paths or a configuration so that the code is not tied to a specific absolute path on one machine.

With this folder scheme and the above components, the project will be well-organized. This layout will make it easier to manage the growing number of files (particularly as we might end up with tens of thousands of XML/PDF files in `data/`) and maintain a **reproducible pipeline** that others can run by following the documented steps.

1 3 For Developers - PMC
https://pmc.ncbi.nlm.nih.gov/tools/developers/

2 4 7 FTP Service - PMC
https://pmc.ncbi.nlm.nih.gov/tools/ftp/

5 OpenRouter vs Ollama: Which AI Runtime Should You Use? | by Ramesh Kannan s | Aug, 2025 | Medium
https://medium.com/@rameshkannanyt0078/openrouter-vs-ollama-which-ai-runtime-should-you-use-83135cf75dd5

6 Best Free AI Models You Can Use on OpenRouter
https://apidog.com/blog/free-ai-models/

8 9 10 11 18 extraction_pipeline.py
file://file-CUWypTApmv1D2iDpzQ51sq

12 How do I get SNOMED CT to download? - NLM Support Center
https://site-bf31cb1c-ac53-4cc2-9ad4-db81f24f361d-gcv.azurewebsites.us/kbArticle/?pn=KA-04087

13 OpenRouter
https://openrouter.ai/

14 15 16 17 Journal of Medical Internet Research - An Entity Extraction Pipeline for Medical Text Records Using Large Language Models: Analytical Study
https://www.jmir.org/2024/1/e54580/

19 20 Folder Structure for Data Analysis | by JJ | Human in a Machine World | Medium
https://medium.com/human-in-a-machine-world/folder-structure-for-data-analysis-62a84949a6ce