ChatGPT

Building a Modular Biomedical Data Extraction Agent

Introduction

Extracting structured biomedical data from literature (e.g. case reports or clinical studies) can greatly accelerate research and data analysis [1] [2]. This guide outlines a comprehensive plan to build a **modular coding agent** for this purpose. The agent will use **free large language models (LLMs)** (via Hugging Face or OpenRouter.ai) and handle diverse document formats to produce **per-patient structured records** according to a user-defined JSON schema. Key challenges addressed include fuzzy biomedical entity normalization (mapping free-text mentions to standard IDs) and continuous self-improvement through prompt tuning and feedback.

We will cover the system architecture, module design, example LLM prompts, required tools/libraries, an implementation checklist, and plans to extend the agent to new domains like patents and clinical trial reports. By following this guide, one can implement a robust Python library (with CLI and Jupyter support) for automated extraction of rich structured data from unstructured biomedical texts.

System Architecture Overview

Figure 1 below illustrates the high-level architecture of the coding agent, highlighting its modular design and self-improvement feedback loop. The system is composed of stages for document ingestion, LLM-based information extraction, ontology mapping, and output evaluation, all orchestrated to produce and continually refine structured outputs.

Figure 1: Flowchart of the agent's architecture, emphasizing the feedback loop for self-improvement. The agent processes text through an LLM with a structured prompt, produces output aligned to the schema, then compares against ground truth (when available) to update an "error memory" and refine future prompts.

The architecture is designed for flexibility and future growth. Documents from various sources are first converted to a uniform text format. The text is then fed into an **LLM extraction pipeline** guided by system prompts that target specific information (genes, phenotypes, treatments, etc.). The raw LLM outputs are post-processed to map entities onto standardized identifiers (HPO terms, DrugBank IDs, HGNC gene symbols, etc.). Finally, if ground truth data (e.g. manually curated CSVs) is available, the agent enters a feedback loop: it evaluates mismatches between its output and the ground truth, and uses those findings to optimize prompts or logic for subsequent runs. This iterative loop enables continuous improvement without model fine-tuning, by honing the prompts and rules that guide the LLM.

Modules and Their Responsibilities

To implement this system, we break it down into distinct modules, each responsible for a part of the process:

- **Document Loader & Preprocessor:** Handles input in various formats (PDF, TXT, HTML, XML, etc.), converting them into plain text. This module may use PDF parsing libraries, HTML/XML parsers, and OCR if needed [3]. It should also split documents by *patient case* if multiple cases are described, using section titles or other cues.
- **Schema & Output Template:** Manages the user-provided JSON schema that defines the structured record format (fields, types, allowed values). This module can generate a template or example output structure to guide the LLM (for in-context learning) and later validate LLM outputs against the schema.
- **LLM Extraction Agent:** The core module that queries one or multiple LLMs to extract information. It constructs **system prompts** for each extraction task (e.g. instructing the model to find all gene mutations, or summarize treatments) and feeds the document text (or per-patient text segment) to the model. It may use different prompts/strategies for different fields but ultimately combines results into the structured record format.
- **Fuzzy Entity Mapper:** This post-processing module takes the LLM's free-text outputs (phenotype descriptions, drug names, gene aliases, etc.) and maps them to standardized identifiers. It handles fuzzy matching using external knowledge – for example, mapping a phenotype description to the closest **Human Phenotype Ontology (HPO)** term, or a drug name to a **DrugBank** entry. It may leverage dictionaries or vector search over ontology definitions to find the best matches [4] [5].
- **Feedback Analyzer:** If a ground truth CSV or annotations are provided for the processed documents, this module compares the agent's output to the true values field-by-field. It computes metrics (accuracy, false negatives/positives per field) and identifies where the model's output was incorrect or missing. This can include parsing differences, incorrect mappings, or hallucinated data.
- **Prompt Optimizer:** Using insights from the feedback analyzer, this module adjusts the LLM prompts or extraction rules. It might append additional instructions, add few-shot examples for fields frequently missed, or simplify prompts that caused confusion. The optimizer could be rule-based (if specific patterns of errors are known) or even use a smaller LLM to suggest prompt improvements. For example, if the agent often misses certain lab values, the prompt can be refined to explicitly ask for those [6] [7].
- **Orchestration & Interface:** The top-level controller that coordinates all modules. It iterates through documents, invokes the loader, runs the LLM extraction and mapping, then calls the feedback loop if applicable. It also provides interfaces: a CLI for batch processing and a Jupyter-friendly API for interactive use. This makes the tool accessible to both command-line users and researchers working in notebooks.

Each module is designed to be **self-contained** and interchangeable. For example, one can update the Document Loader to support a new file format (say, a new XML schema) without affecting other components, or swap in a new LLM in the Extraction Agent if a more capable free model becomes available, thanks to standardized prompt interfaces.

Choosing and Using Free LLMs

A crucial design choice is to use **free, open-access LLMs** to avoid API costs and ensure data privacy. Two recommended sources are Hugging Face (for open model weights) and OpenRouter.ai (for hosted models via a unified API). Notable examples include **DeepSeek V3** and **Mixtral-8x7B**:

- *DeepSeek V3:* A 685B-parameter mixture-of-experts chat model (one of the flagship models on OpenRouter) available in a free variant [8]. Despite its enormous size, it is accessible through OpenRouter with generous limits (e.g. 1000 free messages per day) and has strong performance on complex tasks. DeepSeek V3's free tier can be tapped via OpenRouter's API by obtaining an API key [9].
- *Mixtral-8x7B:* An open-source 45B-parameter Mixture-of-Experts model from Mistral AI (8 experts * 7B each). Mixtral is available on Hugging Face with Apache 2.0 license. It *outperforms Llama2-70B on many benchmarks while being much faster* [10]. This makes Mixtral one of the best cost-performance tradeoff models, even matching or exceeding GPT-3.5 on several tasks [10]. It can be downloaded and run locally (with GPU acceleration) or via Hugging Face's Inference API.

Both models support sufficiently large context windows (e.g. 16k or more tokens) which is important for processing entire papers or lengthy case descriptions. In practice, the agent can abstract a *model interface* such that it's easy to switch between available LLMs. For instance, one can implement a class `LLMClient` with methods `complete(prompt)` that dispatches to the chosen backend (OpenRouter API call for DeepSeek, or local Huggingface transformers pipeline for Mixtral). This abstraction also allows testing different models for quality and speed.

Prompt Design: Using these models effectively requires careful prompt engineering. Because we aim to output JSON or tabular data, it's important to craft the system/user prompts to explicitly request structured output. We can take advantage of in-context learning by showing an example of the desired JSON format in the prompt (especially for models like Mixtral which support few-shot prompting well). According to recent protocols [11] [12], providing step-by-step instructions and examples in prompts can markedly improve accuracy for information extraction tasks. Therefore, our system prompts will often include a brief role instruction (e.g. *"You are an expert biomedical information extractor"*) and guidance like *"Output each patient's data as a JSON object following this schema: ..."*. More detailed examples of prompts are given in a later section.

Document Ingestion and Preprocessing

This agent must handle input documents in **multiple formats** without losing critical information:

- **PDF Documents:** Many papers (especially older ones or those not available as XML) will be PDFs. The Document Loader should use a reliable PDF parsing library to extract text. Libraries like *PDFMiner* or *PyMuPDF (fitz)* are suitable; for example, the ByteScience project built a similar pipeline using PDFMiner to convert PDFs to text [13]. We should be mindful of common PDF parsing issues (headers/footers noise, two-column layouts, special characters). A preprocessing step can clean the raw text, e.g., removing line breaks hyphenating words, filtering reference sections, etc. If a PDF is a scanned image (rare for newer articles), an OCR step (with pytesseract or equivalent) might be needed [3], but assuming mostly digital PDFs is reasonable.

- **Plain Text (TXT):** Easiest case – just read the file. We may still apply cleaning (trim excessive whitespace, ensure consistent spacing around punctuation).
- **HTML:** Some sources might provide HTML content (e.g. web pages or publisher site). For HTML, we can use *BeautifulSoup* to extract visible text and optionally preserve basic structure (paragraphs, headings). Since the agent cares about separating patient cases, using HTML tags (like `<h3>` "Patient 1" headings) can help isolate sections.
- **PMC XML (JATS XML):** Many open-access articles on PubMed Central come in XML format following the JATS schema. We can parse these with Python's `xml.etree` or `lxml` to directly extract sections of interest. For example, find `<sec>` elements with titles "Case" or "Patient". The advantage of XML is that metadata (like tables, figure captions) can be identified and possibly skipped or separately handled. Our loader should be written to handle new XML-based formats as they emerge by adjusting the parsing logic (e.g. if future journals use a slightly different schema).
- **Future Formats:** To future-proof, design the loader such that adding a new format is a matter of implementing a new parser class that conforms to a common interface (e.g. `DocumentParser.parse() -> text`). For instance, if tomorrow there's a JSON format or a new type of database dump, one could plug in a parser for that format. The system could even auto-detect format by file extension or content.

After raw text extraction, **patient segmentation** takes place. Many case reports or case series label patients as "Patient 1, 2, ..." or "Case 1, 2, ...". The preprocessor can split the text on these labels. If a paper is not structured by patient, another strategy is needed (the LLM might then have to distinguish different individuals mentioned, which is harder). But in most rare-disease case series, they do enumerate patients. We can implement simple regex-based splits (e.g. split when a line starts with "Patient" or "Case" followed by a number). Edge cases: ensure that "Patient 10" is not split as "Patient 1" and "0". We can mitigate that by using lookahead/behind in regex or by handling multi-digit numbers. Alternatively, feed the entire text to an LLM asking it to separate sections per patient; however, a rule-based approach is deterministic and preferred for consistency.

The output of this module is cleaned text, optionally divided by patient, ready for extraction.

LLM Prompting Strategy for Extraction

This is the heart of the agent: using LLMs to extract structured info. We propose to break the problem into **sub-tasks with specialized prompts**, as this often yields better results than one giant prompt. Below are key extraction tasks and example system prompts for each:

- **Gene and Mutation Extraction:** We need to find what gene(s) are implicated and what mutations each patient has. A prompt for this could look like:

```
System: You are a medical information extractor. Identify all gene names and
genetic mutations mentioned for the patient. Provide the gene symbols (using
HGNC official symbols) and specific DNA or protein changes if given.
User: [Patient 1's text here]
Assistant: Gene: SLC19A3; Mutations: c.1276_1278delTAC (p.Val426del), c.265A>C
(p.Ser89Arg)
```

In this example, the system prompt instructs the model to output the gene and mutations in a clear format. We might encourage a JSON sub-structure, e.g., `{"gene": "SLC19A3", "mutations": ["c.1276_1278delTAC (V426del)", "c.265A>C (S89R)"]}` for easier downstream parsing. By providing **role hints** ("You are a medical information extractor") and asking specifically for gene symbols, we reduce confusion with other text. Using HGNC official symbols can be aided by giving the model a list of possible gene names (if short) or just relying on its knowledge for common genes.

- **Phenotype (Symptom) Extraction:** Patients often have a list of clinical features. We want to capture those and later map to HPO. A possible approach is to let the LLM output the symptom terms it sees. Prompt example:

```
System: List the clinical phenotypes (signs, symptoms, findings) reported for
this patient in brief form. Use the terms as close to the text as possible.
User: [Patient 1's text here]
Assistant: - ataxia (balance problems)
- ptosis (drooping eyelids)
- metabolic acidosis
- developmental delay (global)
- ...
```

Here the assistant lists key phenotypic terms. We might instruct it further to note negations or absence (e.g., if it says "no seizures", we might want to mark that as an absence of seizures). However, handling negations can be complicated; for the first iteration, we may focus on positive findings only, since the schema might only expect binary presence/absence which we can fill in later. The prompt above tries to ensure the model doesn't miss details by asking for a list of "signs, symptoms, findings".

- **Treatment and Outcome Summarization:** We need to capture treatments given (e.g. "thiamine 100mg/day and biotin 5mg/kg/day") and the outcomes ("improved", "died at 8 months" etc.). Prompt example:

```
System: Summarize any treatments or interventions given to the patient and the
subsequent outcome or response.
User: [Patient 1's text here]
Assistant: Treatment: High-dose thiamine (100 mg/day) and biotin (5 mg/kg/day).
Outcome: Notable clinical improvement after treatment.
```

This prompt directs the model to two pieces of info: what was done, and what happened as a result. In the output, we might later split that into fields (`what_treatment`, `outcome_after_treatment`). The model is usually quite good at summarizing narrative descriptions into concise statements like the above.

- **Demographics and Timeline:** For fields like sex, age of onset, last seen, age of death, etc., those often appear in the text in phrases like "4-month-old girl" (which gives sex=female, age of onset=4 months) or "developed symptoms at age 2". We can craft a prompt focusing on these:

```
System: Extract patient demographics and timeline information: sex, current age
or age at last follow-up, whether alive or deceased (and age of death if
applicable), and age of symptom onset.
User: [Patient 1’s text]
Assistant: Sex: Female; Onset: 4 months; Last seen: 2.5 years old (alive).
```

This instructs the model to output a structured summary of key timeline data. We used terms that align with schema fields (“onset”, “last seen”) to ease mapping. Some of this could also be done with simple regex (for age), but an LLM can interpret context (e.g. if it says “the patient died at 6 months”, the model can fill `alive/dead` status and age_of_death).

Using *separate prompts per sub-task* keeps each query focused. However, note that OpenRouter models like DeepSeek may have a cost in latency for multiple calls. If needed, we can combine some prompts or use a single prompt that asks for a JSON with multiple sections. For example, a single prompt could say: *“Provide a JSON with fields: sex, age_of_onset, gene, mutations, phenotypes, treatment, outcome…”*. The model might output a single JSON object per patient. This is elegant when it works, but if the model makes an error in one part, it could affect the entire JSON. A compromise approach is to do it in steps (as above) and then assemble the final JSON.

Few-Shot Examples: Especially for smaller open models (Mixtral) that benefit from demonstration, we should include an example of input text and expected JSON output in the prompt (one that is not from the test set). This acts as a guide. For instance, we could use a short dummy case: “*A 10-year-old boy with XYZ1 gene mutation presented with ataxia and seizures. He was treated with DrugX and is currently stable.*” and show a JSON mapping those. According to LLM best practices, this will improve output formatting [14] [6]. We should ensure the example follows the schema exactly (e.g., uses correct enum values like “m”/“f” for sex, etc.).

Finally, the LLM extraction module should incorporate **error handling**: if the model’s output is not parseable (e.g., it outputs a narrative paragraph instead of JSON), the agent can detect that and retry with a more stern prompt (e.g. “Output JSON only, no explanations”). Some prompt iterations may be needed initially to find what works best with our chosen model. This is part of prompt optimization, which we discuss next.

Fuzzy Entity Mapping and Ontology Integration

After the LLM provides the extracted data (still in text form, e.g. gene names, symptom strings), we need to normalize these to standard identifiers: - **Phenotypes to HPO:** The Human Phenotype Ontology is a standardized vocabulary of phenotypic abnormalities [15]. Our schema lists many HPO terms (e.g. HP: 0001251 for “ataxia”). For each phenotype mention extracted, the agent should find the closest matching HPO term. A straightforward way is to maintain a dictionary of known HPO names/synonyms to IDs. We can populate this by loading the HPO ontology (e.g., using the `pronto` or `owlready2` library to read the OWL/OBO file). For fuzzy matching, one could use string similarity (Levenshtein distance or via the `rapidfuzz` library) or a vector-based approach. A **vector search** approach might yield better semantic matching: for example, embed the extracted phrase and all HPO term names in the same embedding space and find the nearest neighbor. Tools like *FAISS* can do efficient nearest-neighbor lookup on embeddings. In fact, recent research showed that augmenting an LLM with an HPO term retriever improved normalization accuracy from ~62% to 90% [4]. Our design can incorporate a similar *retriever*: for each symptom string, retrieve top candidate HPO terms either by lexical fuzzy match or embedding similarity. Then either automatically pick the top candidate if above a similarity threshold, or (future improvement) prompt an LLM (like GPT-4 or our current model if capable) with those candidates to choose the best fit [5] [16]. Initially, a simpler rule-based mapping might suffice given many symptoms will have exact or near-exact name matches in HPO (the CSV header suggests terms like "ataxia", "ptosis" which directly map). We must also handle situations like the schema expecting a binary or coded value for presence/absence. For example, "ataxia (1=Symptom, 2=never acquired)" indicates we should output "1" if ataxia is present, "2" if the patient never achieved the ability (like never walked, implying severe ataxia), or blank/0 if not mentioned. The agent's logic can determine this from context: if text says "she never learned to walk" that implies "ataxia = 2" (never acquired walking skill). Such nuanced logic might be implemented with custom rules for those specific fields in the schema. - **Drugs to DrugBank IDs:** DrugBank is a comprehensive database of drugs. Each drug has a name and a DrugBank accession (DB####). We likely won't have the model output the DrugBank ID directly (it almost certainly won't know random IDs). Instead, we can map the drug name. DrugBank provides a downloadable vocabulary (or one can use PubChem or RxNorm as alternatives for name-to-identifier mapping). A simple approach: maintain a dictionary of drug name to DrugBank ID for common drugs of interest, or use an API like UniChem or RxNorm to find IDs. However, since our focus is more on extracting the drug name and perhaps class, we might not need the actual ID in output, depending on schema. (The schema provided doesn't explicitly list a drug ID field, just "what treatment" as free text – but the requirements asked for mapping drugs to DrugBank, so likely we should plan to include an ID or at least ensure standardized drug names). For fuzzy matching drug names, we can use case-insensitive matching and also account for synonyms (DrugBank has many brand names etc., which complicates it). This could be a future enhancement; for now, even just outputting the drug name as in text is an OK starting point, then mapping can be added later. - **Genes to HGNC:** Similarly, genes should be standardized to official HGNC symbols. If the paper uses an old name or a nucleotide accession, we should convert to gene symbol. In practice, many papers already use official symbols in all-caps (e.g. *SLC25A19*). We could use the HGNC complete gene list (available as a CSV/TSV from HGNC) to verify and correct symbols (for instance, if the model extracted "Slc19a3", we can uppercase it). If a gene isn't found or looks like a typo, we might flag it for manual review. Additionally, if multiple gene fields exist (gene, gene1, gene2 in schema), the agent should distribute them appropriately (e.g. if a patient has two genes mutated, fill both fields). - **Other Entities:** The schema has many specific lab values and imaging findings. We might treat those similarly: e.g., if the LLM extracts "elevated lactate", we map "elevated" to the code in the schema (perhaps "e" for elevated). This is more of a straightforward lookup (the schema enumerations like `"unnamed_60": enum ["e","l","n"]` presumably stand for elevated, low, normal). We can implement a mapping where we see words like "elevated" or "high" and assign `e`, "low/decreased" -> `l`, "normal" -> `n`. This will likely be rule-based in code rather than expecting the LLM to output the single-letter code.

By structuring the entity normalization as a distinct module, we ensure the LLM's job is simpler (just find and output info in natural language), and our deterministic code or lightweight AI handles the standardization. This approach is commonly recommended to combine the strengths of LLMs (reading comprehension) with knowledge bases [5].

Feedback Loop and Prompt Optimization

One of the standout features of this agent is its ability to learn from its mistakes without model re-training. The **feedback loop** uses a ground truth (manually processed data in CSV) to identify errors in extraction, and then the **prompt optimization module** adjusts accordingly. Here's how to implement and utilize these:

Ground Truth Comparison: After generating structured output for a document, if a reference CSV row is available (for example, the user might have a gold-standard extraction for that paper, as given in `manually_processed.csv`), the agent should compare each field. This can be done field by field: - If a field is numeric or categorical, check if the values match exactly. - If a field is textual (e.g. a mutation description), some tolerance might be needed (minor formatting differences might be acceptable, but major content differences are errors). - For fields that are lists (like multiple symptoms), one could compare sets or do precision/recall of the items.

The output of this comparison could be a report or even just a structured object listing mismatches. For example: *Patient 1: mismatch in* `gene_2` *(expected SLC19A3, got null)* or `dysmorphic_features` *expected "microcephaly" not found*. Automating this comparison is possible: treat both outputs and ground truth as JSON and diff them. The **Feedback Analyzer** essentially implements this.

Analyzing Errors: Not all mismatches are the LLM's fault – some could be ground truth issues or ambiguous text. The agent can apply simple heuristics to categorize mismatches: - "LLM output missing info present in ground truth" – likely an extraction miss. - "LLM output has extra info not in ground truth" – possibly hallucination or extracting something not annotated by human (could be an error or could be something the human missed!). In clinical IE, it's known that sometimes models pick up details overlooked in annotation [17]. Those instances might require human review of ground truth. - "Value mismatch" (LLM got something but it's incorrect) – e.g., output age of onset 5m vs truth 4m. That's a detail error.

For each category, we have a different mitigation: - Missing info: likely we need to **make the prompt more explicit or add an example** for that field. For instance, if the model often misses "dysmorphic features", we might add to the prompt: *"Also note any dysmorphic features (physical anomalies) if mentioned."* The LLM-AIx protocol suggests that if information is present but model didn't extract, one should provide a more detailed explanation or few-shot example in the prompt [18] [19]. - Hallucinated/extra info: instruct the model to be concise and stick to the text. We might need to emphasize: *"If a piece of data is not stated, do not infer it."* For example, if the model assumed a patient was male because of name but it wasn't stated, that's a hallucination. We can refine the prompt to warn against that. If hallucinations persist, consider using a smaller temperature or a more extractive approach (some models allow an extraction mode). - Incorrect values: this might mean the model misinterpreted something. For example, if text says "no ataxia" and the model put ataxia = 1 (present), that's a misunderstanding of negation. To fix, we might adjust the prompt to handle negations, or add a rule in post-processing (if you see "no X", then X=absent). Prompt-wise, we can include a line: *"If the text explicitly negates a symptom, that symptom should be marked as absent."* Another kind of value error could be mixing up patients if the text wasn't properly segmented – in that case, ensure segmentation is working (maybe the prompt accidentally included two patients' info at once).

Incorporating Feedback: Once we identify the needed prompt changes, the Prompt Optimizer module can apply them. This can be as straightforward as maintaining a set of *prompt adjustment rules*: - e.g., *if gene field was wrong => ensure next prompt explicitly asks for gene.* But likely we always ask for gene anyway. A more nuanced rule: *if the model missed an HPO term that was in text, add that term to the example prompt if possible.* Over iterations, one could build a library of tricky terms and ensure the prompt covers them. - The optimization can also involve trying a different prompt format or splitting tasks differently if we notice a pattern of errors. For instance, if the combined prompt approach (JSON all at once) doesn't perform well, the feedback might suggest switching to the multi-prompt approach described earlier. Conversely, if multi-step prompts lead to some info being lost between steps, maybe try a single-shot prompt.

In an automated setting, we could implement a loop that runs on a set of labeled documents: for each, compare output to truth, then automatically tweak prompts and test again. This starts bordering on *reinforcement learning via prompts*. However, a simpler approach is iterative manual refinement with the help of metrics: - After initial run on a few examples, identify the worst-performing fields. - Tweak the prompt or add an example focusing on those. - Re-run and see if metrics improve for those fields.

The **error memory** concept (as illustrated in Figure 1) refers to storing known problematic cases so the system doesn't repeat mistakes. For example, if it consistently mis-labels "elevated lactate" as normal, we keep that memory and ensure future prompts or mappings catch it. We could store a small JSON of known errors (like `"elevated lactate" -> ensure classify as 'e'`).

To validate that these improvements are working, the Feedback Analyzer can compute precision, recall, F1 for each field over a test set [20]. This provides a quantitative way to track progress as we refine the system.

One should also consider that sometimes the *ground truth itself might need revision* [21]. If the agent consistently outputs something that seems logical but differs from the CSV, it's possible the human annotators made an error or interpreted differently. In a development setting, this might prompt reviewing the source. The agent could even highlight such cases ("LLM extracted info not in ground truth") which might reveal incomplete annotations [19].

In summary, the feedback loop turns our pipeline into a living system: as more papers are processed and occasionally manually verified, the agent's prompts and rules get better, leading to higher accuracy on subsequent extractions. This approach leverages *in-context learning* of LLMs instead of fine-tuning weights, which is efficient and keeps the system adaptable [2] [12].

Packaging as a Python Library (CLI & Jupyter Interfaces)

We will implement the above modules in a Python package for ease of use and distribution. Key considerations:

- **Modular Code Structure:** Create a Python package with submodules like `ingestion.py`, `extraction.py`, `mapping.py`, `feedback.py`, etc., corresponding to the major components. Each submodule can contain classes or functions for that part of the pipeline. For instance, `ingestion.py` might have a `DocumentLoader` class with methods `load_pdf()`, `load_html()`, etc., and `ExtractionAgent` in `extraction.py` handles LLM calls. Using a clear namespace makes the library easier to maintain.
- **Command-Line Interface (CLI):** Provide a CLI (using `argparse` or `click`) that allows users to run the agent on documents without writing code. For example, after installing, a user could run:

  ```
  extract_biodata --model deepseek-v3 --schema schema.json input_papers/
  output.csv
  ```

  This would load all papers in `input_papers` (mix of PDFs, etc.), run the extraction, and save combined CSV output. CLI options can include model selection, enabling/disabling the feedback loop (if ground truth provided), etc. By making the CLI intuitive, even non-programmers can utilize the tool (similar to how LLM-AIx provides an easy UI [22]).

- **Jupyter/Notebook Usage:** Notebooks are popular for medical researchers. We should ensure our library can be imported and used in an interactive way. This means functions should return data in Python objects (e.g. a list of patient record dicts) for further analysis. We can provide example notebooks demonstrating usage, such as loading a paper, displaying the extracted JSON, and maybe visualizing which fields were found. The design should avoid requiring heavy CLI-only constructs when used as a library (e.g., if using `argparse`, guard it under `if __name__ == "__main__"` so it doesn't run on import).

- **Installation and Dependencies:** List needed packages (for PDF parsing, for OpenRouter API calls, etc.) in a `requirements.txt` or `setup.py`. We should pin versions that are known to work. Also, consider the environment – some users might not have GPU: in such cases, they might rely on OpenRouter models or smaller models. We can make the model selection such that if a local model name is given, it tries to load it with transformers (requiring enough hardware), whereas if an OpenRouter name is given, it uses an API call. Document these usage patterns clearly.

- **Logging and Debugging:** Include logging to track what the agent is doing (e.g., "Parsed 3 patients from PDF X", "Calling LLM for gene extraction on Patient 2..."). If something goes wrong or performance is bad, these logs help pinpoint the step. For integration in other systems, we might allow turning off verbose logging.

By packaging it well, others can extend the library. For example, a user could write a new module to handle patent documents and plug it in (we will discuss patents next). The library should also allow partial use – e.g., maybe someone only wants the phenotype extractor. If they import our `ExtractionAgent` and feed it text, they should get structured output easily. This means designing the classes with optional components (like you can skip feedback if no truth available, etc.).

Implementation Checklist (Step-by-Step)

Below is a step-by-step **TODO list** for implementing the agent, summarizing the tasks in a logical order:

1. **Project Setup:** Initialize a Python project (e.g. via Poetry or pip) and create the basic package structure. Install key libraries: `pytesseract` (for OCR, optional), `PyMuPDF` or `pdfminer.six`, `beautifulsoup4`, `lxml`, `transformers` (for local LLMs), an HTTP library for OpenRouter calls, `faiss-cpu` (for vector search), etc.
2. **Load JSON Schema:** Read the user-provided JSON schema (e.g. `table_schema.json`). You can use Python's `json` module to load it as a dict. This schema will drive what fields to extract. Optionally, create a Python class (e.g. `PatientRecord`) that mirrors the schema fields for easier handling.
3. **Implement Document Loader Module:** Develop functions to handle each format:
4. `parse_pdf(file_path) -> str` using PDF library,
5. `parse_txt(file_path) -> str`,
6. `parse_html(file_path) -> str`,
7. `parse_xml(file_path) -> str`. Integrate them in a single `load_document(file_path) -> str` that picks the right parser based on file extension or content. Include any cleaning steps (unnecessary newlines, etc.). Test on a sample PDF and XML to ensure text is extracted correctly (compare a few lines to original).

8. **Implement Patient Segmentation:** In the loader or a separate step, split the text by patient case if multiple. For initial simplicity, look for headings like “Patient” or “Case”. If found, split and label segments. If not, assume the whole text is one patient or a general description.
9. **Set Up LLM Access:** Write a class for model inference. E.g. `LLMClient` with methods:
10. `complete(prompt:str) -> str` for single-turn completion. If using OpenRouter: the `complete` method will call the OpenRouter API endpoint with the prompt and retrieve the model’s output. If using local (Hugging Face), load the model and tokenizer (taking care with device placement if GPU available) and generate. Ensure you load in inference mode (half precision if possible for speed). Test this with a simple prompt to verify model is responding.
11. **Draft Initial Prompts:** Create template prompts for each sub-task (gene extraction, phenotype list, etc.) possibly stored in a dictionary or separate text files for easy editing. Include placeholders for inserting the patient text. For example: `prompts["gene"] = "Extract gene and mutation: {text}"`. Use the schema to decide which prompts you need – likely we group some fields logically (demographics, genetics, symptoms, treatment/outcome).
12. **Implement ExtractionAgent:** This will orchestrate calling the LLM for each needed piece and assembling the results. Pseudocode for one patient:

```
data = {}
# demographics
prompt = prompts["demo"].format(text=patient_text)
demo_output = llm.complete(prompt)
data.update(parse_demo_output(demo_output))
# gene
prompt = prompts["gene"].format(text=patient_text)
gene_out = llm.complete(prompt)
data.update(parse_gene_output(gene_out))
# phenotypes
prompt = prompts["pheno"].format(text=patient_text)
pheno_out = llm.complete(prompt)
data.update(parse_pheno_output(pheno_out))
# treatment/outcome
prompt = prompts["treatment"].format(text=patient_text)
treat_out = llm.complete(prompt)
data.update(parse_treatment_output(treat_out))
```

Here, `parse_*_output` are helper functions to clean or structure the raw LLM text. For example, `parse_pheno_output` might split bullet points into a list. Or ideally, if the prompt made the LLM output JSON, we can directly `json.loads` it. In practice, it might be safer to let the LLM output plaintext and parse ourselves, since JSON might be malformed. We should be prepared to handle slight variations (e.g. different ways of phrasing).
13. **Implement Fuzzy Mapping:** After assembling `data` for a patient (with values as extracted text), run the mapping logic:
14. For each phenotype field, map text to HPO code or to the schema format (like setting that symptom’s column to 1 if present). Possibly pre-load a list/dict of HPO terms. You might also simply check if the phenotype appears in the schema (the CSV header lists many HPO terms we expect). For each known term in schema, see if it was mentioned in the LLM’s phenotype list; if yes, mark it present.

15. For gene names, verify against HGNC list. If the gene symbol is known, great. If not, consider if it's a gene alias – you could have a map of aliases to symbols from HGNC data.
16. Normalize drug names casing (e.g., "Prednisone" vs "prednisone") and optionally look up DrugBank ID via a prepared dictionary.
17. Map qualitative lab results to coded values as needed (if the LLM output "lactate was high", set the corresponding field to "e"). This step will likely be a series of `if/else` or dictionary lookups for each relevant field.
18. **Validate & Format Output:** Now convert the `data` dict to the final structure (JSON or CSV row). If JSON output is desired, ensure it conforms to the schema (all required fields present, types correct). Use the schema definitions to, for example, cast numbers to float/int, or set default values for missing. If CSV output is needed, output fields in the schema order. This is where a direct JSON-to-CSV conversion can happen since schema is known.
19. **Implement Feedback Evaluation:** If ground truth CSV is provided for the input document(s), implement a comparison. You could load the CSV into a list of dicts keyed by patient ID or number. After extraction, compare each field. For automation, compute simple accuracy metrics: e.g., `accuracy = (# of fields exactly matching / total # of fields)`. More granularly, track per-field accuracy. If doing multiple documents, accumulate stats across them. This module can print a summary table of, say, field name, % matched, and notable errors. Optionally, log mismatches in detail for debugging.
20. **Prompt Optimization Rules:** Based on the common mismatches observed, encode some improvement rules. For instance:
    - If many patients miss a certain symptom, consider adding a hint in the phenotype prompt like "(e.g., any ataxia, seizures, etc?)".
    - If the LLM often outputs a narrative where we wanted keywords, modify the system prompt to be more explicit ("Use bullet points" or "one keyword per line").
    - If the output JSON has formatting issues, perhaps add `` \n```json `` in the prompt to coerce JSON format (some models follow that). Maintain these adjustments so that they can be applied iteratively. This might simply mean editing the prompt templates and re-running extraction to see improvement. We can incorporate an automated loop to test new prompt versions on the ground truth set.
21. **Command-Line Interface:** Write a `cli.py` that uses argparse to accept input paths, model selection, etc. The CLI should call the above modules in sequence. Example flow in CLI:
    1. Load schema.
    2. For each input file: load text, segment patients.
    3. For each patient: run ExtractionAgent -> get `data` -> map to structured output.
    4. Append to results list.
    5. After all, if `--truth` provided, run feedback evaluation on results vs truth CSV.
    6. Save output CSV or JSON as requested. Provide helpful console output during the run (how many patients processed, any warnings).
22. **Testing on Example Document:** Before deploying, test the whole pipeline on a known example (like the provided PMID32679198 case series). Compare the output to the manually_processed.csv. Identify where it diverges. This will likely reveal areas to refine prompts or mapping rules. Iteratively fix those as per the feedback loop strategy. This step is crucial to ensure the system works end-to-end.
23. **Documentation and Examples:** Write a README explaining usage, and perhaps provide a Jupyter notebook demonstrating extraction on a sample text. Include example prompts (some from above)

in the documentation for clarity. Also mention the need for appropriate hardware if using local models (and provide alternatives like OpenRouter for those without GPUs).

Following this checklist will result in a functional agent that can be continuously improved. Each step can be tracked and checked off to ensure completeness.

Plan for Future Expansion (Patents and Clinical Trials)

The modular nature of our agent makes it relatively straightforward to extend to new document types like **patent texts** and **clinical trial records**:

- **Patents:** Biomedical patents (for drugs, devices, etc.) are typically long documents with technical language and often multiple examples or case data. Key differences: patents may not be focused on individual patients, but some contain case studies or experimental results on subjects. To handle patents, we might need:
- A new *DocumentLoader parser* for patents if they come in a unique format (often PDF too, but could be HTML from patent office). Parsing might need to remove boilerplate sections (legal text, claims) and focus on the "Examples" or "Description" section where experimental results are described.
- If extracting patient data from patents, likely it's about participants in a study or case reports included in the patent. The schema might need to expand or adjust for patents (they might include different info like patent ID, invention details, etc.). But if our goal remains extracting biomedical measurements and outcomes, the existing framework suffices.
- The LLM prompts might need to be tuned to patent phrasing. Patents often use formal language and sometimes non-standard terms. The agent may benefit from a specialized model for patents or at least some few-shot examples using patent excerpts.
- Patents are often very long (tens of thousands of words). Our system may need to chunk them (perhaps process section by section). We should plan to integrate a *document splitter* for extremely long texts, or use an LLM with very large context window if available.

- We should also be cautious of different units or terminologies in patents; the ontology mapper might need additional entries (for example, a patent might refer to a compound by a code instead of a generic name, making drug mapping harder).

- **Clinical Trials:** By this, we refer either to documents describing clinical trial results (like journal articles or reports) or entries from trial registries (like ClinicalTrials.gov).

- If dealing with trial registry data (which is often semi-structured XML/JSON already), the task is more about mapping that data to our schema. It might require a different ingestion approach (parsing the registry fields). However, many registry fields align with a structured schema already (e.g. patient demographics, outcomes). We could write a converter rather than use LLMs for those, since they are structured.
- For journal articles about clinical trials (e.g. a paper on a Phase II study of a drug), these often have aggregate data rather than per-patient data. Our agent as designed is more for individual patient records. To handle trial reports, we might shift focus to extracting *cohort characteristics* and outcomes. That's a slightly different schema (statistics, not individual entries). It might be better served by a different mode of the agent or a new schema. In future, we can incorporate multiple schema types and have the user specify which to use.

- If the aim is to extract something like adverse events or specific results from trial reports, the LLM prompts would be adjusted accordingly (e.g. “Extract the number of patients with outcome X in the treatment group vs placebo group”). This becomes more of a data mining task than case record extraction. We can still apply similar prompt techniques but the JSON schema would be different.
- Ontology mapping for trials might involve other vocabularies (like conditions to MeSH or interventions to a taxonomy). We can extend the mapper with additional ontology dictionaries as needed.

- **Scaling and Performance:** For both patents and trials, the volume of data might be larger (patent databases are huge, and trial registries have tens of thousands of entries). We should consider performance optimization:

- Perhaps integrate a **vector database** to store embedded text of documents for faster search within or across documents (especially if doing something like question answering from them).
- Use batch processing for LLM calls if possible (some libraries allow sending multiple prompts in parallel to utilize GPUs fully or to not over-hit rate limits).
- Possibly incorporate smaller models for initial tasks (like identifying which sections of a patent contain patient data, using a classifier model, then only run the big LLM on those sections).

Adapting the Feedback Loop: As we extend to new document types, the feedback mechanism also extends. For each new domain, we’d accumulate some ground truth examples (perhaps manually annotated patents or trial summaries) and then refine prompts. The agent could maintain separate prompt optimizations per domain, if needed, since what works for academic articles might differ from patents.

In conclusion, the architecture we built – with clearly separated parsing, extraction, and normalization stages – is well-suited to expansion. Adding support for patents and clinical trials would primarily involve **adding new parsers and tweaking the prompt strategy**, rather than redesigning the whole system. This confirms the value of the modular approach. With future integration of domain-specific models (e.g., a fine-tuned model on patent text or a specialized clinical trials Q&A model), the agent can become even more powerful while following the same overall pipeline.

Conclusion

We have outlined a detailed plan for constructing a modular coding agent that transforms unstructured biomedical texts into structured per-patient data records. By leveraging free LLMs (such as DeepSeek and Mixtral) for their language understanding, and combining them with rule-based post-processing and a feedback-driven prompt refinement loop, the system can achieve accurate and scalable information extraction [11] [12]. The guide covered the full spectrum from document ingestion to output evaluation, providing example prompts and technical pointers along the way.

The resulting Python library will enable researchers to quickly obtain structured datasets from papers, improving the efficiency of systematic reviews, meta-analyses, and database building in the biomedical domain. Continual learning via prompt optimization ensures that the agent gets better with use, adapting to new data and domains. We started with scientific articles, but the design is ready to tackle patents, clinical trial reports, and more, making it a long-term asset for biomedical text mining efforts.

Finally, remember that the success of such an agent depends on rigorous testing and iteration. Each corpus (be it rare disease case reports or pharma patents) might bring surprises, but with the modular tools and strategies described, one can methodically address them. Happy coding, and may this agent accelerate your biomedical data extraction tasks!

Sources: The approach and considerations above draw on recent work in applying LLMs to medical text extraction [23] [3], ontology-based normalization techniques [4] [5], and the capabilities of new open-source LLMs [8] [10]. These informed our design to ensure the agent is built on state-of-the-art foundations.

---

[1] [2] [3] [6] [7] [11] [12] [14] [17] [18] [19] [20] [21] [22] [23] LLM-AIx: An open source pipeline for Information Extraction from unstructured medical text based on privacy preserving Large Language Models - PMC
https://pmc.ncbi.nlm.nih.gov/articles/PMC11398444/

[4] [5] [16] A Simplified Retriever to Improve Accuracy of Phenotype Normalizations by Large Language Models
https://arxiv.org/html/2409.13744v1

[8] [9] OpenRouter
https://openrouter.ai/deepseek/deepseek-chat-v3-0324:free

[10] Mixtral
https://huggingface.co/docs/transformers/en/model_doc/mixtral

[13] ByteScience: Bridging Unstructured Scientific Literature and Structured Data with Auto Fine-tuned Large Language Model in Token Granularity
https://arxiv.org/html/2411.12000v1

[15] HPOLabeler: improving prediction of human protein–phenotype ...
https://academic.oup.com/bioinformatics/article/36/14/4180/5831828