ChatGPT

Automating Patient-Level Data Extraction from Leigh Syndrome Literature

Architecture Diagram and Rationale

Figure: System architecture with four coordinated tracks (A–D) for data extraction, review, browsing, and agent orchestration.

The system architecture is composed of four integrated tracks, labeled **A** through **D**, working in concert to replicate clinician-curated tables from literature and enrich them with ontology mappings. **Track A** focuses on *patient-level data extraction* from research papers. **Track B** handles *systematic review automation* including query parsing and PRISMA-style reporting. **Track C** provides a *patent and clinical trials explorer* (a Lens-like module for intellectual property and trial data). **Track D** is the *agent orchestration layer* that coordinates the autonomous agents (via frameworks like AutoGen and Agent Zero) and LLMs (using Ollama for local models). This architecture ensures that literature data is comprehensively ingested and transformed into a structured knowledge base, which can then be queried and analyzed in flexible ways.

Track A: Patient-Level Data Extraction Pipeline

Track A ingests open-access **PubMed/PMC literature** on Leigh syndrome and extracts structured patient data. It begins with a **metadata retrieval** step (using PubMed E-Utilities or similar APIs) to gather article metadata (PMIDs, DOIs, titles, etc.) for all relevant case reports and series. Next, for each article, the system fetches the **full-text PDF** (or XML) and parses it using **GROBID**, a machine learning library for extracting and structuring scientific PDFs [1]. GROBID quickly converts each paper into a well-formed XML/TEI document, capturing title, authors, abstract, body sections, and references [2] [3]. This structured output allows the system to locate sections like case descriptions, methods, and tables with high fidelity.

Once the text is extracted, the pipeline identifies any **embedded tables** that contain patient data (demographics, clinical findings, labs, etc.). For robust table extraction, the system uses **Camelot**, a Python library designed to accurately extract tabular data from PDFs. Camelot can handle a variety of table structures (using lattice or stream parsing) and outputs each table as a structured DataFrame. This is critical for capturing any clinician-generated tables in the papers that summarize patient cohorts. If a PDF page has complex formatting or scanned images, the system flags it for manual review or uses OCR as a fallback, but primarily relies on text-based PDF parsing (Camelot requires text-based tables).

After obtaining raw text and table data, the pipeline performs **concept extraction and enrichment**. An *LLM-based NLP agent* (leveraging a local model via Ollama or an API) scans the text to identify key patient-level facts. These include: patient identifiers (within the paper), demographics (sex, age), genetic variants, clinical phenotypes, lab results, treatments, and outcomes. The agent uses prompt-based instructions to output these facts in a structured JSON or tabular format for each patient. It is guided by the schema of the manual table (with ~290 fields covering various domains). For example, if a case report mentions *"the patient developed ataxia at 2 years and had elevated lactate"*, the extractor will record `ataxia = true`

`(onset 2 years)` and the lactate level in the appropriate fields. The extraction agent is aware of context (e.g. differentiating multiple patients in one paper) and uses the paper's structure to reset context between cases as needed.

Next, the extracted terms are **mapped to biomedical ontologies** and normalized. This involves linking *phenotypes* to HPO codes, diseases to MONDO or OMIM identifiers, genetic variants to standard notations, lab tests to LOINC codes, and medications to RxNorm (details in the Ontology Integration section). By normalizing terms, the system enriches the raw data – for instance, mapping "failure to thrive" to HPO: 0001508 and standardizing "Leigh syndrome" to MONDO:0019497 – which facilitates cross-paper aggregation and query. At this stage, the system may call out to ontology lookup services or use local dictionaries to get the codes (e.g., using the HPO JSON database for phenotypes). Any ambiguous mapping or unknown term is flagged for human review.

Finally, Track A stores the cleaned, structured data into the **Knowledge Base**. This consists of a relational database (PostgreSQL) for the main patient table and a graph database (Neo4j) to capture relationships (e.g. linking patients to genes, phenotypes, publications). In addition, a **vector database (Weaviate)** indexes textual embeddings of case descriptions and abstracts for semantic search. Weaviate is an open-source vector DB that enables semantic similarity queries by comparing embeddings [4], which helps find similar cases based on narrative context. The knowledge base thus supports both exact queries (via SQL/graph on structured fields) and fuzzy searches (via vector similarity). At the end of Track A, we have a **comprehensive patient database** extracted from the literature – essentially replicating and enhancing the manual annotation table provided, but now automatically populated. This database is the foundation for the other tracks and can be updated continuously as new papers are published.

Track B: Systematic Review Module

Track B automates the process of performing a systematic literature review on Leigh syndrome using the data gathered in Track A. It accepts **user queries or review topics** (for example, *"Compare clinical features of SURF1-related Leigh syndrome across reported cases"* or *"What is the efficacy of certain interventions in Leigh syndrome patients?"*). The process begins with **query parsing and understanding** using an LLM. The *Query Parser Agent* interprets the user's question and formulates a search or filter strategy. For instance, it might determine that the query is focusing on a subset of patients (e.g., those with a specific gene mutation or symptom) and thus derive inclusion criteria.

Using those criteria, the system performs **evidence retrieval** against the knowledge base. This involves querying the structured database (Postgres/Neo4j) for patients or papers matching the criteria (e.g., filter all patients where `gene = SURF1` and extract their phenotypes and outcomes). The retrieval can also leverage the vector index for semantic matching – e.g., find cases with similar descriptions if the query is qualitative. The result is a set of relevant patient records and references. Because Track A pre-ingested and indexed the literature, this step is extremely fast compared to a fresh PubMed search. It essentially retrieves *all evidence from the already-collected corpus* that answers the question.

Next, the module handles **PRISMA-style screening and reporting**. The system knows the total number of sources in the database (e.g., all Leigh syndrome case reports). It can report how many were *considered* and how many *met the inclusion criteria* for the query. For example, if our database has 250 case reports and the query was about SURF1 mutations, maybe 50 of those contain SURF1 patients – the system would generate a PRISMA flow summary: *250 records identified, 250 screened (no additional sources needed), 200 excluded (not*

SURF1), 50 studies included for analysis. Automating PRISMA ensures transparency in how data was filtered and is aligned with reporting guidelines [5]. The system can output a PRISMA flow diagram or at least textually describe these numbers for the user's reference.

Finally, the **summary report generation** step compiles the findings. An LLM-based *Report Writer Agent* (nicknamed the "STORM Report Writer") takes the filtered data and produces a narrative synthesis. It might create tables on the fly (e.g. comparing frequency of phenotypes among the included cases) and cite the relevant papers. For example, it could write: "Among the 50 SURF1-related Leigh syndrome patients, the most common features were hypotonia (80%) and developmental regression (70%) [5]. Mortality by age 3 was reported in 60% of cases [5]. Thiamine or vitamin therapy was associated with stabilization in 5 cases [5]." (Citations here would correspond to the literature sources via the knowledge base). The agent ensures to reference appropriately and may attach the list of included PMIDs. The resulting output to the user is akin to a mini-review on the query, with evidence-backed statements and even the ability to output data in table form if requested. This fulfills the role of a **live systematic review** that can be re-run anytime the database updates, saving countless hours of manual literature review.

Track C: Patent and Trial Explorer (Lens-like Functionality)

Track C extends the system's capabilities beyond published case reports into the domains of **patents and clinical trials**. This module allows the user (or an agent on their behalf) to query intellectual property databases and clinical trial registries for information related to Leigh syndrome (or associated genes, therapies, etc.), much like the Lens.org platform integrates scholarly and patent data. The user can issue a query such as *"patents related to gene therapy for Leigh syndrome"* or *"ongoing clinical trials for mitochondrial diseases"*.

First, the query is processed by an LLM for **entity and keyword extraction**. The system identifies key terms (gene names, disease terms, drug names) in the query to construct effective search strings. For instance, if the query mentions *gene therapy*, the agent might look for specific gene targets like "MT-ATP6" or "SURF1" in patent text. If it's a broad query for trials, it may identify terms like "Leigh syndrome", "phase II", "treatment" to search the ClinicalTrials.gov API.

The module then performs a **patent search** and a **clinical trials search** in parallel. For patents, it can leverage an API or dataset (such as Lens's patent search API) to find patents matching the query terms. For example, it might search patent titles/abstracts for "Leigh syndrome" or related scientific terms. For clinical trials, it queries databases like ClinicalTrials.gov or WHO ICTRP for studies related to Leigh syndrome or mitochondrial disorders. Both searches return a list of relevant records (patent documents or trial registrations). The system may apply filters (e.g., trials with results vs. ongoing, patents in the last 10 years, etc.) depending on the query parameters.

After retrieval, the results from both patents and trials are fed into a **synthesis and reporting** stage. An LLM agent summarizes the findings: for patents, it might list key inventions or therapies and their assignees; for clinical trials, it might summarize how many trials are ongoing, and highlight any interventions under study. For instance, the output could be: "**Patents:** 3 patents were found on gene therapy for Leigh syndrome, including one by University X on AAV-mediated *SURF1* delivery (Patent US123456) and two on new mitochondrial antioxidants. **Clinical Trials:** 5 trials are registered, with 2 active: one testing a high-fat diet in Leigh syndrome (NCT....) and another on a new drug Y (NCT....). Preliminary

results from a completed trial of dichloroacetate showed modest benefits." The agent provides links or identifiers (patent numbers, NCT IDs) for the user to follow up.

Track C thus acts as a **research assistant for translational insights**, ensuring that the system not only aggregates published literature (Track A/B) but also keeps an eye on emerging therapies and patented innovations. This is particularly useful for clinicians or researchers who want to know the *broader landscape* – including if someone has patented a therapy approach or if patients can be enrolled in trials. The design is "Lens-like" in that it presents a unified interface to explore scholarly knowledge (from the database) alongside patents and trials, offering a more comprehensive view of the field.

Track D: Multi-Agent Orchestration and LLM Engine

Track D is the coordination layer that ties everything together using **autonomous AI agents** and large language models. The system employs an *agentic framework* (such as Microsoft's AutoGen library or the Agent Zero framework) to manage the complex workflow across Tracks A, B, and C. **AutoGen** provides a high-level framework for defining multiple agents that can converse and cooperate to accomplish tasks [6]. In our system, the orchestrator (Agent 0) delegates subtasks to specialized sub-agents and collates their results. **Agent Zero** similarly introduces a hierarchy where a top-level Agent 0 (with the user as its superior) can spawn child agents for subtasks. This design keeps each agent's context focused and prevents one agent from being overwhelmed by the entire workflow.

Concretely, we define several roles for the agents in Track D:

- **Master Orchestrator Agent (Agent 0)** – This agent is the central brain that receives user requests and breaks them into subtasks. It decides, for example, that a request for "summarize case findings" will involve using Track B's systematic review process, or that a new literature update triggers Track A ingestion. The orchestrator communicates with sub-agents by issuing them instructions and can integrate their outputs. It ensures the overall task (e.g., "Build the Leigh syndrome patient table") is completed end-to-end by coordinating the team.

- **Metadata Triage Agent** – This agent handles incoming literature metadata. For example, if new PubMed records are found or a large list of PMIDs is provided, the Metadata Triage Agent filters out irrelevant ones (e.g., non-case-report articles) and prioritizes which PDFs to fetch first. It may use simple heuristics or an LLM ("Given titles/abstracts, which papers likely contain patient-level data?"). This ensures Track A focuses on the most relevant sources. It can mark each paper with tags (case report, review, animal study, etc.) based on metadata.

- **Concept Extractor Agent** – This agent is responsible for reading the full text (or relevant sections) and pulling out the raw patient information. It might operate per paper or per patient. The agent's prompt (system message) instructs it to output specific fields (e.g., *"Extract patient demographics, symptoms (with onset age), lab results, etc. from the following text."*). It may call smaller utility functions as tools (like regex for specific formats or invoking Camelot for table portions). The Concept Extractor populates a structured data draft for each patient.

- **Table Mapper Agent** – Once the concept extractor has a draft of patient data, the Table Mapper Agent aligns it to the canonical schema (the master table with 294 columns). It ensures that each extracted piece goes into the right field and uses consistent units and coding. For instance, if the

extractor output “elevated lactate of 3 mmol/L”, the mapper knows to put that under “serum lactate” and also compute lactate/pyruvate ratio if pyruvate is available. It converts free-text into coded form where possible (e.g., “moderate hypotonia” -> mark the hypotonia field as present). Essentially, this agent transforms narrative data into a *clean row* in the database, handling any format normalization.

- **Validator Agent** – The Validator checks the output rows for completeness and consistency. It might cross-verify data points: e.g., if a patient is marked as alive (0=alive) but has an age of death, that’s a conflict to flag. Or if HPO codes for “ataxia” and “gait disturbance” are both false, but text mentioned “ataxic gait”, it might catch that something was missed. The Validator uses rules and possibly another LLM pass to ensure quality. Any uncertainties or low-confidence extractions are flagged for human review, which form part of the user to-do list (e.g., “Patient 5 in PMID 123456 – unclear mutation description, please verify manually”).

- **STORM Report Writer Agent** – This agent focuses on composing human-readable outputs, whether it’s the summary of findings for Track B or drafting sections of a manuscript. *STORM* (Stanford’s AI tool for writing comprehensive articles) is integrated here to automatically generate reports or even full papers. Given structured data or query results, the STORM writer agent can create formatted text with appropriate context. For example, after Track A builds the database, the user could ask for a “digital case series report,” and this agent would use the data to write a draft publication with introduction, methods (describing how data was gathered), results (the table of patients and analysis), and discussion. It essentially automates writing tasks that otherwise require synthesizing multiple data points into narrative form.

- **Digital Twin Composer Agent** – Looking towards more advanced usage, this agent uses the aggregated patient data to create *simulated patient avatars* or disease models. The agent might interface with disease simulation software or simply collate all data for a given genotype into a comprehensive profile (a “digital twin” of the typical patient). It can answer *“what-if” scenarios* (e.g., how might a patient with mutation X progress, based on past cases?) by drawing from the natural history data in the knowledge graph. In the present system, this agent might generate a summary per patient (a digital profile) or per subtype of Leigh syndrome. In the future, it could plug into predictive models to simulate outcomes under various interventions (see Future Plans).

All these agents operate under the orchestration of **Track D**, which ensures that complex tasks are broken down and executed efficiently. The agents communicate through a shared memory or messages (the exact mechanism depends on the framework: AutoGen provides a conversation-based coordination [6], while Agent Zero uses a hierarchical delegation model). We use **Ollama** to host local LLMs for these agents’ reasoning to ensure data privacy and offline capability. Ollama is an open-source tool allowing large language models to run on local hardware [7], which is ideal for an academic server or a Mac with Apple Silicon. By using local models (which can be domain-tuned if needed), we avoid reliance on external APIs and can better control the model behavior for each agent’s role.

In summary, Track D provides the *adaptive intelligence* of the system. It dynamically manages the flow: for example, when a new PDF arrives, the orchestrator might say *“Metadata Agent, verify this is relevant. If yes, Extractor Agent, process it. Then Table Mapper, update the DB, and Validator, check the entry.”* When a user poses a complex query, the orchestrator might engage multiple agents: *“Parser Agent, figure out what they want. Retriever Agent, get the data. Writer Agent, draft an answer.”* This design makes the system **modular, scalable, and fault-tolerant** – each agent can be improved or replaced independently (for instance, swapping in a

more advanced concept extraction model later) without disrupting the whole. The rationale for this multi-agent approach is to mirror the workflow of a human research team (one person gathering papers, another extracting data, another writing the report) and to leverage specialization, which in practice leads to better accuracy and manageability than one giant monolithic AI trying to do everything at once.

Technology Stack and Justification

To implement the above architecture, we leverage a robust technology stack, picking the best tool for each job. Below is an overview of the key components and why they were chosen:

- **Literature Parsing – GROBID:** We use GROBID (GeneRation Of BIbliographic Data) to convert PDFs of papers into structured text. GROBID is highly efficient and produces rich TEI XML with sections, references, and more [2]. Its ability to extract detailed metadata (titles, authors, etc.) and body text with minimal errors is essential for downstream NLP. Running as a local REST service (via Docker) allows batch processing of hundreds of PDFs quickly. GROBID's structured output also simplifies locating specific sections (e.g., case descriptions) programmatically.

- **Table Extraction – Camelot:** Camelot is a Python library specialized in PDF table extraction, which we include for capturing any tabular patient data [8]. It offers both *lattice* mode (good for tables with drawn cell lines) and *stream* mode (for tables with whitespace separation), giving flexibility to handle various journal formats. We chose Camelot for its accuracy and ease of integration (pandas DataFrame outputs), as well as its open-source nature which allows customization if needed. Since many clinical papers present key data in tables, Camelot ensures we don't miss those.

- **Database – PostgreSQL and Neo4j:** For structured data storage, we selected PostgreSQL as a reliable relational database. It will store the core patient table (each row representing a patient with columns for each attribute). SQL is useful for performing rigorous filtering, aggregations, and ensuring data integrity (with constraints and schema). However, relationships between entities (patients, their genes, papers, phenotypes) are better represented in a graph database. We integrate Neo4j to create a *knowledge graph* of Leigh syndrome cases: nodes for patients, diagnoses, phenotypes (HPO terms), genes, etc., connected by relationships (e.g., Patient–HAS_PHENOTYPE→Ataxia). Neo4j excels at traversing such relationships (e.g., find all patients connected to a particular HPO term or gene). The combination of SQL + graph DB gives us both structured query capabilities and semantic querying for relationships.

- **Vector Index – Weaviate:** We include Weaviate as a vector database to enable semantic search over unstructured text (like case descriptions and abstracts). Weaviate is an AI-native, open-source vector DB with built-in semantic search and hybrid search capabilities [4]. By storing embedding vectors of texts, the system can answer similarity questions (e.g., "find patients most similar to this new case description") that SQL or Neo4j alone cannot. Weaviate's APIs and scaling ability (billions of vectors) ensure we can expand the corpus in the future without performance loss. It complements the exact matching of the graph/SQL with fuzzy matching powered by language understanding.

- **Ontologies and Knowledge Sources:** We use standard biomedical ontologies to normalize and enrich data. The **Human Phenotype Ontology (HPO)** provides codes for phenotypic abnormalities, ensuring that symptoms are consistently represented. **Monarch Disease Ontology (MONDO)** is

used for disease entities; MONDO merges multiple disease vocabularies into a coherent ontology, giving us a stable ID for Leigh syndrome and related disorders. We also reference **SNOMED CT**, a comprehensive clinical terminology, to cover general medical concepts and ensure compatibility with electronic health records (SNOMED CT is the most comprehensive multilingual healthcare terminology). For lab tests and measurements, we adopt **LOINC** codes, since LOINC provides universal identifiers for laboratory tests and clinical observations (e.g., there's a specific LOINC code for "Lactate [Moles/volume] in CSF"). Medications and treatments are mapped to **RxNorm**, the standard nomenclature for clinical drugs, to handle drug names and dosages. These ontologies are integrated via lookup services or local files; we plan to use resources like the OBO Foundry and BioPortal for reference data. By using established ontologies, we enhance interoperability and enable cross-dataset analyses (e.g., an HPO-coded phenotype can be linked to other databases of rare diseases).

- **LLM Backbone – Ollama + LLM Models:** For all language processing tasks (parsing text, answering queries, writing summaries), we rely on large language models. We deploy these through **Ollama**, which lets us run LLMs on local hardware (particularly suited for Mac M1/M2/M3 or a Linux GPU server). Ollama is an open-source tool allowing direct local execution of models without needing internet access [7]. This choice is motivated by data privacy (patient case data remains local) and cost (no API fees). We can choose different models for different tasks: for example, a GPT-4 class model (or Llama 2 70B) for complex reasoning in the Orchestrator, a specialized medical model for concept extraction (like a BioGPT or a fine-tuned Llama on clinical text), and a code-oriented model for any code generation tasks. These models are containerized by Ollama, making it easy to switch or update them. The **tech stack also includes GPU support** (if on an academic server with a GPU) to speed up model inference. The system requirements are moderate: a single GPU and >16GB RAM will suffice for medium-sized models, while Apple Silicon Macs can run 30B+ parameter models with accelerated performance.

- **Agent Orchestration – AutoGen, Agent-0, and Crew:** To build the multi-agent ecosystem, we leverage existing frameworks. Microsoft's **AutoGen** library allows defining agents and tools in Python with a conversation-based paradigm, making it straightforward to implement the agents and their interactions [6]. It supports asynchronous, event-driven messaging which is useful for the complex workflow (agents can work in parallel when needed). **Agent Zero** (Agent-0) offers a different but complementary approach, emphasizing a persistent, self-improving agent that can spawn sub-agents for tasks. We take inspiration from Agent Zero for the hierarchical delegation and long-term learning – for instance, the Master agent could learn from repeated tasks to optimize prompts or recall past errors (Agent Zero's memory feature would allow that). **Crew** (CrewAI) is another toolkit focusing on multi-agent task structuring; we consider it for high-level flow design and as an interface to define flows in a no-code fashion. CrewAI provides templates (or "flows") for agents to follow, which might accelerate development of, say, a chain that goes from PDF to database automatically. In summary, the tech stack uses AutoGen for low-level agent management and message passing, Agent Zero principles for agent design (self-reflection and tool creation), and CrewAI resources for best practices in multi-agent workflow design. This combination yields a flexible yet powerful orchestration layer without having to reinvent the wheel.

- **Miscellaneous Tools:** Additional tools in the stack include Python libraries such as **Biopython/ Entrez** for fetching PubMed records, **PyMuPDF or PDFminer** as backup PDF text extractors for GROBID (in case some PDFs fail or for quick text extraction where full structure isn't needed), **Pandas**

for data manipulation and analysis of the extracted table, and **NetworkX** or **Cypher queries** for graph analytics on Neo4j. For deployment and environment, we use **Docker** extensively: GROBID runs in a Docker container, Neo4j and Weaviate have official Docker images (which we can link via Docker Compose), and even Ollama can be run as a service. This containerization ensures reproducibility and easy setup on new machines. We also plan to use **Jupyter notebooks** during development for prototyping extraction logic and verifying outputs, which then get translated into the production pipeline code.

Each component of the tech stack was chosen for its reliability, community support, and compatibility with an academic/open-source ecosystem. The system is built mostly in **Python**, given its rich ecosystem for scientific computing and NLP, and the fact that most of these tools (GROBID's client, Camelot, AutoGen, etc.) have Python interfaces. The technology choices are justified by the need to balance **accuracy**, **performance**, and **maintainability** for a project that must handle non-trivial NLP on scientific text and produce clinically relevant structured data. By combining these technologies, we create a system that can be deployed on a single GPU server or even a high-end laptop, and which can be maintained/extended by other researchers familiar with standard Python data stacks.

Ontology Integration for Clinical Data

A core design principle of this system is to map all extracted data to **standard biomedical ontologies**. This not only ensures consistency within our database but also allows linking and comparing our data with external data sources (like clinical databases, other case series, etc.). Below we detail how different categories of data are standardized using ontologies and controlled vocabularies:

- **Phenotypes (Symptoms and Signs):** We use the **Human Phenotype Ontology (HPO)** as the backbone for coding patient phenotypic abnormalities. HPO provides over 18,000 terms describing human phenotypes and is widely used in rare disease research. In the manual table, each clinical feature was already associated with an HPO term (e.g., *ataxia* – HP:0001251, *optic atrophy* – HP:0000648). The system's extraction agent, whenever it identifies a symptom, will assign the corresponding HPO ID. We maintain a mapping dictionary of common symptom phrases to HPO terms (including synonyms). For example, "failure to thrive" is mapped to HP:0001508, "developmental delay" to HP:0001263, and so on. If an extracted symptom doesn't directly match our dictionary, we can query the HPO ontology (via the OBO API or a local lookup) to find the best match. In cases where a very specific term isn't in HPO, we may fall back to a parent term. All phenotype fields in the database store both a boolean/presence and the HPO code for positive findings. This allows for easy counting of phenotype frequencies and also advanced queries like "find all patients with any neurological abnormality" by leveraging HPO's hierarchy (since HPO terms are in a directed acyclic graph, we can use ancestor terms to group related phenotypes).

- **Diseases and Diagnosis:** Leigh syndrome itself, and any related disorders mentioned, are coded using **MONDO** (Monarch Disease Ontology) and cross-referenced with **OMIM** and **Orphanet** where applicable. Leigh syndrome's MONDO ID is MONDO:0019497 (which aggregates OMIM 256000, ORPHA 778, etc.). If a paper mentions a broader category like "mitochondrial disease" or a specific subtype like "X-linked Leigh syndrome", we attempt to capture that. MONDO is a comprehensive merged ontology of diseases, so it's our primary choice for disease labels. Additionally, since clinicians often use **ICD-10** codes, we can map Leigh syndrome to ICD-10 G31.82 (for Leigh's disease) in the background for any outputs that might require ICD coding. For differential diagnoses or

comorbid conditions mentioned, we also prefer MONDO/OMIM codes. The database can store multiple diagnosis codes per patient if needed (e.g., some patients have “Leigh syndrome” plus an etiological diagnosis like “SURF1-related Leigh” which might be represented as separate entries). Using standardized disease codes allows potential integration with hospital records or other datasets in the future and helps avoid ambiguity (e.g., “Leigh-like syndrome” might be coded as Leigh syndrome in MONDO plus a modifier).

- **Genetic Variants:** Leigh syndrome can be caused by numerous genetic mutations (both mitochondrial DNA and nuclear DNA). We capture gene information in standardized form: gene names are normalized to **HGNC official symbols** (e.g., *MT-ATP6*, *SURF1*, *PDHA1*). The “mt or nDNA” column in the table indicates mitochondrial vs nuclear genome, which we keep. For variants, we store them in HGVS notation if available (e.g., m.8993T>G for the MT-ATP6 mutation, or c.845_846del for a nuclear gene variant). We may integrate with **ClinVar** or **dbSNP** to get reference IDs for common variants, but that is optional. Ontologically, genes aren’t part of HPO/MONDO, but we can link them via **OMIM gene entries** or **Gene Ontology (GO)** terms if discussing pathways. At minimum, each patient’s causative gene is recorded and can be cross-referenced to external knowledge (like “all SURF1 patients”). We also note zygosity (homozygous, heterozygous) and inheritance patterns in standardized terms (like using HPO terms for inheritance, e.g., HP:0000006 for autosomal recessive).

- **Laboratory and Imaging Data:** Lab test results (lactate, pyruvate, etc.) are mapped to **LOINC** codes for standardization. For example, CSF lactate might use LOINC code 2524-7 (“Lactate [Moles/volume] in Cerebral spinal fluid”). Using LOINC ensures that if we compare our data to other clinical data, we know we’re referring to the same test. Likewise, blood lactate could be 2526-2 (Lactate [Moles/volume] in Blood). We normalize units as well (mMol/L typically for lactate, etc.). Imaging findings from MRI are more free-form in text, but we try to map them to terms from **RadLex** or use HPO where possible (HPO has terms like “Basal ganglia lesion” or “Leukodystrophy” etc., which can describe MRI findings). For instance, MRI basal ganglia lesions could map to HPO:0002134 (abnormality of the basal ganglia). If RadLex (the radiology ontology) is used, it would be for very detailed imaging descriptors; however, to keep things unified, we might stick to HPO and simple yes/no fields (as the table has columns for each region’s MRI involvement).

- **Treatment and Interventions:** We record treatments (e.g., thiamine, biotin, dichloroacetate, respiratory support) and map these to standard identifiers when possible. **RxNorm** provides normalized drug names and codes, so we use that for medications (e.g., “Thiamine” has an RxNorm ID, likewise “Coenzyme Q10”). For interventions like dietary changes or physical therapy, there may not be a specific ontology term; in such cases we store descriptive terms but could tag them with SNOMED CT codes if available (SNOMED CT has many procedure and therapy codes). For example, “Ketogenic diet” has a SNOMED code, “Mechanical ventilation” has a code, etc. Using SNOMED CT (which covers procedures and devices) complements RxNorm for non-drug interventions and ensures consistency with clinical terminologies. The outcome of treatments (improvement/stable/progression) are standardized to a small vocabulary (we could use terms from Clinical Global Impression or just a 3-point scale internally).

- **Clinical Course and Outcomes:** Many of these concepts can also be mapped. “Age of onset” and developmental milestones can be linked to HPO terms like “Global developmental delay (HP:0001263)” or “Developmental regression (HP:0002376)” if those occurred. We capture whether the

patient is alive or deceased (the table uses 0/1). Death can be coded with a concept like "HP:0001636 Premature death" if needed, or simply handled as a separate field with age. We also record "last seen" (last follow-up age), which isn't an ontology term but is important metadata. If cause of death is mentioned, we could code that (e.g., respiratory failure – which is HPO:0002878 if a phenotype, or perhaps ICD-10 code for cause of death if available). We additionally ensure that any mention of complications (like "cardiomyopathy", "renal tubulopathy") are tagged with HPO or appropriate codes so that these can be queried systematically.

- **Ontology Storage and Cross-Referencing:** In the knowledge graph (Neo4j), phenotype nodes are labeled with HPO IDs and include the name and perhaps definition. Disease nodes have MONDO and also xrefs like OMIM. By storing the ontology IDs, we can perform graph queries like "find all patients who have any phenotype that is a child of 'Movement disorder (HP:0100022)'" using the ontology relationships (we might load the HPO structure into the graph as well, or query an API on the fly). For now, we keep it simpler: direct annotations on patients, and we rely on external tools for hierarchical queries if needed. We maintain a reference table mapping each HPO code we use to its name and perhaps a category (neurologic, metabolic, etc.) for easier grouping in analyses.

In summary, the ontology integration ensures that our *extracted data is semantically interoperable*. A phenotype like "ataxia" isn't just a string – it's linked to an HPO entry that places it in context (a neurological symptom, part of Ataxia and Cerebellar signs category). This allows powerful queries and also makes the data suitable for feeding into other tools, such as the phenotype matching algorithms of the Monarch Initiative or phenotype-driven differential diagnosis tools. It also facilitates **data validation** – e.g., if an extracted term cannot be mapped to any known ontology term, that's a sign of a potential extraction error or a new term requiring curation. By leaning on these widely accepted standards (HPO, MONDO, SNOMED, RxNorm, LOINC, etc.), our system's outputs can readily contribute to collective knowledge about Leigh syndrome and can integrate with clinical IT systems if needed (for example, exporting the patient data in a FHIR format with coded entries).

Deployment Guide (Step-by-Step)

This section serves as a comprehensive guide to deploying the system, from hardware setup to data ingestion and processing. We assume an environment either on a **single-GPU server (Linux)** or a **Mac (Apple Silicon)**, as these are typical academic setups. Key steps and considerations are outlined below:

1. System Requirements and Environment Setup

- **Hardware:** At minimum, the system requires a machine with a modern CPU, 16 GB+ RAM, and sufficient storage (the PDF corpus and databases will take a few GB). For LLM tasks, having a **GPU** (NVIDIA with CUDA support, 8+ GB VRAM) is recommended to speed up model inference. If a GPU is not available, the system can run on CPU with smaller models or using Apple Silicon optimization on Mac (M1/M2 chips have a 16-core Neural Engine that Ollama can use).

- **Operating System:** Linux (Ubuntu 20.04/22.04 LTS or similar) or macOS (Monterey or later). Windows is not explicitly tested, but using WSL2 or Docker containers should work if needed.

- **Python Environment:** Install Python 3.10 or higher (for compatibility with AutoGen and other new libraries). We suggest setting up a virtual environment (using `venv` or `conda`). Ensure pip is up to

date. If on Mac M1/M2, use `miniforge` or similar to get arm64 native Python for best performance.

- **Basic Dependencies:** It's helpful to install system packages for PDF processing: e.g., Poppler (for PDF to text if needed), Ghostscript (required by Camelot's image-based extraction), and Java (for GROBID). On Ubuntu, you'd do: `sudo apt-get install poppler-utils ghostscript default-jre`. On macOS, Homebrew can be used: `brew install poppler ghostscript openjdk`.

2. Installing Required Tools and Libraries

- **GROBID:** The easiest way is to use the Docker image. Install Docker if not already installed. Then pull the GROBID image:

```
docker pull grobid/grobid:0.7.2
```

Run the container (adjusting version if needed):

```
docker run -d --name grobid -p 8070:8070 grobid/grobid:0.7.2
```

This launches GROBID's RESTful service at `http://localhost:8070`. Verify by visiting `http://localhost:8070/api/isalive`. If Docker is not preferred, alternatively build from source (cloning the GitHub repo and running with Gradle) [9] [10], but Docker is simpler.

- **PostgreSQL and Neo4j:** We recommend using Docker for these as well, for isolation. For Postgres:

```
docker run -d --name postgres -e POSTGRES_PASSWORD=yourpassword -p 5432:
5432 postgres:15
```

This starts Postgres 15 with default user `postgres`. For Neo4j:

```
docker run -d --name neo4j -p 7474:7474 -p 7687:7687 -e NEO4J_AUTH=neo4j/
adminpassword neo4j:5
```

Open Neo4j's browser at `http://localhost:7474` and login with `neo4j/adminpassword` (or your set password) to ensure it's running. We will later define the schema (nodes and relations) once data is ready.

- **Weaviate:** If using Weaviate for vector search, you can deploy a Weaviate instance via Docker similarly:

```
docker run -d --name weaviate -p 8080:8080 semitechnologies/weaviate:latest
```

Weaviate can run in-memory for small data; for larger scale, connect it to a proper database or disk. Keep default settings (it has a console at port 8080 you can test). Alternatively, if you prefer not to set up Weaviate initially, you can skip vector search; but if you do, you'll need Python's `weaviate-client` to ingest data later.

- **Python Libraries:** With the Python environment activated, install necessary libraries via pip:

```
pip install grobid-client camelot-py[cv] biopython pandas openpyxl neo4j
weaviate-client
pip install autogen agent0 crewai  # hypothetical package names if
available
pip install transformers accelerate torch  # for LLMs if not using Ollama
```

- `grobid-client` might refer to a simple wrapper or we can use `requests` to call GROBID's API (the guide above [11] [12] shows how to post PDFs).
- `camelot-py[cv]` installs Camelot with OpenCV support (needed for the image-based table parsing).
- `biopython` gives us Entrez for PubMed queries.
- `openpyxl` if you want to read/write Excel (for verifying with the provided manual table).
- `neo4j` is Neo4j Python driver for inserting/querying graph data.
- `weaviate-client` for connecting to Weaviate.
- AutoGen/Agent0: If Microsoft AutoGen is on pip, it might be `pip install autogen` (or install from GitHub). Agent0 might require pip install from its GitHub (`pip install agent-zero` if available). CrewAI might not have a pip package; it could be a set of examples to follow or install via `pip install crew` if it exists – this part may be optional as it's more for orchestration scaffolding.
- `transformers` and `torch`: if using local HuggingFace models or needed for certain LLM tasks (Ollama might abstract this away if using its runtime).

Additionally, install any other needed packages: `pip install PyMuPDF pylatexenc` (for alternative PDF parsing or if processing LaTeX in text), and `networkx` if doing graph analysis in Python.

- **Ollama (for LLMs):** On Mac, install via Homebrew:

```
brew install ollama
```

On Linux, currently Ollama has builds for Ubuntu; you can download a binary or use Docker. If not using Ollama, you can skip and instead plan to use HuggingFace `transformers` to load a model. If Ollama is installed, verify by running `ollama --version`. Then fetch a model:

```
ollama pull llama2:13b
```

(As an example pulling Llama-2 13B model. Ensure you have the license if needed.) The model will download and be ready to serve. You can test run:

```
ollama run llama2:13b --prompt "Hello"
```

This should output a greeting from the model, indicating it works. By default, Ollama runs a local server at `localhost:11434` for programmatic access.

- **AutoGen/Agent Framework Setup:** If using AutoGen, after installing, you might want to quickly configure a simple example to ensure it's working (AutoGen might require OpenAI API keys for some examples; but our usage will point it to local models). We will configure it in code, so no separate service needed – it's a library that helps manage agents within our Python application.

3. Metadata Fetching from PubMed

With the environment ready, the first data task is to fetch all relevant literature metadata from PubMed. Leigh syndrome being a rare disease, we want to capture: - All case reports and case series on Leigh syndrome (including those not explicitly labeled as such but describing Leigh or Leigh-like cases). - Possibly relevant clinical studies or trials publications, though the focus is on case-level data.

PubMed Query: We can use a PubMed query like `"Leigh Disease"[MeSH] OR "Leigh syndrome"[tiab] OR "Leigh-like syndrome"[tiab]` to retrieve references. In practice, the user provided a CSV of ~705 references (likely obtained via such queries and some filtering). To replicate: - Use Biopython Entrez or the `pymed` library to search. For example:

```
from Bio import Entrez
Entrez.email = "your.email@domain.com"  # NCBI requires an email
Entrez.api_key = "YOUR_NCBI_API_KEY"  # recommended to avoid limits
handle = Entrez.esearch(db="pubmed", term="Leigh syndrome AND case report",
retmax=10000)
record = Entrez.read(handle)
pmids = record['IdList']
```

This returns PMIDs for articles matching the query. You might refine the query to exclude unrelated things; for example, ensure "case report" is in the text or use filters (PubMed has Publication Type filter for Case Reports). - Once we have PMIDs, use Entrez.efetch to get details:

```
ids = ",".join(pmids)
handle = Entrez.efetch(db="pubmed", id=ids, rettype="Medline", retmode="text")
records = Medline.parse(handle)
```

For each record, we can extract Title, Journal, Authors, PublicationType (to identify case reports vs reviews), DOI, and if available, the PMCID (PubMed Central ID). The provided CSV already has a column for `PMC` links, which suggests using **PMC** to get full text for open-access articles. We can also query the PMC OAI-PMH service or NCBI's `efetch` for PMC to retrieve the open access subset.

- **Filtering:** Some references might be general reviews or animal studies. We can filter by PublicationType containing "Case Reports" or look at MeSH terms (if "Case Reports" is a MeSH term assigned). The CSV shows entries where PubType includes "Journal Article, Review" – those might be multi-type or narrative reviews. We likely exclude pure reviews from patient data extraction but might keep them in the list for completeness. The Metadata Triage Agent can later decide to skip extraction on those.

- **Storing Metadata:** Save the fetched metadata to a file or directly to the database. For example, create a table in Postgres for articles with fields: PMID, title, journal, year, authors (as text or a separate related table), doi, pmcid, pub_type, etc. Or simply keep it in memory as we proceed to fetching PDFs.

4. Retrieving Full-Text Articles and Supplements

For each article identified: - **Via PMCID:** If an article has a PMCID (as seen in the CSV, many do), it means the full text is in PubMed Central (often as Open Access). We can use the PMCID to download the XML or PDF. For example, NCBI provides a direct FTP link to XML for Open Access: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMCXXXXXX/` returns the HTML/XML. Also, `https://www.ncbi.nlm.nih.gov/pmc/utils/oa/oa.fcgi?id=PMCXXXXX` can give links to PDF or XML. We might prefer PDF to feed to GROBID, since GROBID is trained on PDF layout. But if XML is available, we could bypass GROBID for that article and parse the XML directly. However, not all PMCs have nicely structured XML (some older ones might, but many do).

- **Via DOI/Publisher:** If no PMCID is available (article not open access in PMC), we have to rely on other means. This might involve using institutional access to get the PDF. If we cannot automatically download due to paywalls, those might need to be handled manually (or skipped if focusing on open access only). Another approach: check if the article is available via services like **Unpaywall** or find an author manuscript version. For this deployment, assume we stick to the open-access ones (which are numerous for rare disease case reports).

- **Downloading PDFs:** For each article to be ingested, download the PDF file to a local directory (e.g., `data/pdfs/PMID_<id>.pdf`). Ensure file naming is clear (using PMID or PMCID). We can script this: if we have a URL (like those PMC links in CSV, which actually are to HTML pages), often adding `format=pdf` or replacing part of URL yields the PDF. For example, `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1234567/pdf/XYZ.pdf` sometimes is the pattern. If not straightforward, using `requests` to fetch the page and parse for a PDF link is an approach. Alternatively, use `selenium` or a headless browser for tricky cases. Given a large number of articles, it's worth doing this systematically with error handling and rate limiting (NCBI is generally okay with automated download of OA content, but still be polite).

- **Supplements:** Some case series might have supplementary files (like detailed tables, genetic data, etc.). If identified (in PMC XML or publisher site), those should be downloaded too. Often PMC XML

has `<supplementary-material>` tags with URLs. Save those (e.g., as PDF or Excel). We can later feed them to Camelot or parse accordingly if they contain data.

- Keep track of any article that fails to download, and possibly mark it for manual retrieval later.

5. Processing Full Text with GROBID and Camelot

Once PDFs are in hand: - **Start GROBID** (if not already running from earlier Docker step). - Use the Python requests approach or a GROBID Python client to process each PDF. For example:

```
import requests
files = {'input': open(pdf_path, 'rb')}
res = requests.post("http://localhost:8070/api/processFulltextDocument",
files=files, params={'teiCoordinates': 'biblStruct'})
tei_xml = res.text
```

This sends the PDF and gets back TEI XML (optionally with coordinates for references if needed). Save this XML to a file (for record or debugging) and then parse it (e.g., using ElementTree or lxml). GROBID's TEI will have `<text><body><div>` sections, etc. We might try to identify where case details are. Often case reports have sections titled "Case Report", "Case Presentation", or in multi-patient series they may have "Patient 1", "Case 2", etc. The TEI `<head>` tags give section titles, and `<p>` tags the paragraphs. For multi-patient papers, it might be necessary to split the text by patient – sometimes authors label them or number them. This can be tricky and might need heuristic or agent help. But at least we have all text content extracted.

- After GROBID, we run **Camelot** on the PDF (or on specific pages) to extract tables. Camelot can be run like:

  ```
  import camelot
  tables = camelot.read_pdf(pdf_path, pages='all')
  ```

  It will return a list of tables. We should inspect if any table looks like the type of clinical summary we need. We could use Camelot's metrics (accuracy, whitespace) to guess which tables are meaningful. Alternatively, look at captions or number of rows. Many case report papers might not have extensive tables. But if they do (like a table listing all patients in a series), that table is gold – it likely mirrors what we want. Camelot outputs can be converted to DataFrame (`tables[i].df`). We can then either integrate that directly (with some cleaning) or cross-check against our own extraction to validate. We will store extracted tables in a temporary list. Each cell is text; we might later parse those (for example, a table cell might contain "F/2y (deceased at 3y)" which we'd parse into sex, onset age, outcome, etc.).

- **Integration of GROBID and Camelot outputs:** The Orchestrator/Extraction Agent will have to combine these. A strategy: feed the textual content and any table content together to the Concept Extractor Agent. Possibly format a prompt like:

```
[Section: Case Presentation]
Patient 1: ... (text) ...
Table 1: Clinical summary of patients.
| ID | Sex | Age onset | ... | Phenotypes | ... |
| 1  | M   | 6m        | ... | hypotonia, seizures | ... |
...
```

This prompt would allow the LLM to use both narrative and table to extract data. If doing it purely programmatically: we can convert table data to a dictionary and merge with text-extracted data (the Validator agent can later reconcile discrepancies).

- **Storing Intermediate Data:** For each paper, after processing, we can store a JSON with what we got: metadata (from earlier), text (maybe not whole text but key sections), tables data, etc. This can be useful if we want to rerun extraction without parsing PDFs again. Also, this is a point to do basic text cleaning (remove references tags, newlines, etc. that might confuse the LLM).

6. Running the Extraction and Mapping Agents

With the content prepared, we invoke our *Extraction pipeline*. This can be done in a loop for each paper, or using the agent framework to parallelize if resources allow (e.g., spawn multiple extractor agents for different papers). Key steps each agent or function will perform: - **Identify Patients in Text:** If a paper has multiple patients, the agent should split the text accordingly. Heuristics: look for "Patient 1:" or the patient initials, or numbering. If none, assume one patient. If GROBID output separates `<div>` for each case (not guaranteed, but some structured abstracts do), use that. Otherwise, the agent might need to read the whole and figure it out. We could also rely on the manual count from the table (if present in the text, like "In this series of 3 patients, ..."). - **Extract Fields:** For each patient identified, the agent will pull out all relevant fields. This can be done by prompt:

```
You are a medical information extraction agent. Extract the following fields for
each patient: Sex, Year of birth, Age of onset, Age at last follow-up or death,
Genetic mutation and gene, Key phenotypes, Lab results (lactate, etc.), MRI
findings, Treatments given, Outcome.
Output as JSON.
```

The agent then outputs JSON which we parse. (We ensure to instruct it to keep formats standardized, e.g., use numeric ages in months/years, use lists of HPO codes for phenotypes if possible). - **Map to Ontologies:** After raw extraction, another function or the same agent can call a mapping function to replace phenotype text with HPO codes (using our dictionary). Similarly for labs (e.g., "lactate 5 (high)" – we know normal range so we mark it as elevated and store value 5 mmol/L under the correct field). We can incorporate this in the prompt, but likely safer to do via post-processing functions for determinism. For example:

```
phenos = extracted["phenotypes_list"]
mapped_phenos = []
for ph in phenos:
    code = hpo_map.get(ph.lower())
```

```
    if code:
        mapped_phenos.append(code)
    else:
        mapped_phenos.append("UNKNOWN")
extracted["phenotypes_hpo"] = mapped_phenos
```

We do similar for other categories (if a drug name in treatments, look up RxNorm using an API or local list). - **Populate Database:** Using the PostgreSQL connection, insert the patient data as a new row in the patient table. If the patient is from a new paper, also ensure the paper's info is in a publications table (with PMID as key). Link them via PMID foreign key. For Neo4j, create nodes and relations: e.g., a Patient node with ID (maybe compound of PMID and patient number), relations like (Patient)-[:HAS_PHENOTYPE]->(Phenotype term) for each HPO, (Patient)-[:HAS_GENOTYPE]->(Gene), etc. Many of these nodes (Phenotype, Gene) will be reused between patients, so check if node exists before creating a new one (use MERGE in Cypher to avoid duplicates). For example:

```
MERGE (p:Patient {id: "PMID12345_P1", sex:"F", onset_months:6, death_age:
36, ...})
MERGE (g:Gene {symbol:"SURF1"})
MERGE (p)-[:HAS_GENE]->(g)
MERGE (phen:Phenotype {hpo:"HP:0001252", name:"Hypotonia"})
MERGE (p)-[:HAS_PHENOTYPE]->(phen)
...
MERGE (d:Disease {mondo:"MONDO:0019497", name:"Leigh syndrome"})
MERGE (p)-[:DIAGNOSIS]->(d)
MERGE (pub:Publication {pmid:12345})
MERGE (p)-[:REPORTED_IN]->(pub)
```

This way the graph gradually grows with each patient and we can query it later.

- **Validation:** The Validator Agent or code runs some checks on the inserted data. This can involve running predefined SQL queries or logic, for example:
    - Check required fields: if any crucial field is blank (sex, onset age, gene), log a warning.
    - Check logical consistency: if `alive=0` (alive) but `Age of death` is filled, mark conflict.
    - Check phenotype conflicts: if "never achieved sitting" is indicated (which implies motor delay) but motor milestones are marked normal, etc. Some of these might be complex to automate; the agent could attempt to infer contradictions by reading the patient summary again with a prompt like *"Given these extracted data, do they contradict each other or the text?"*. Likely, simple rule-based checks suffice for now. Any flagged issues can be written to a log or a separate table (Flag table listing patient ID and issue). The user can later address these.
- This process repeats for all papers. It's wise to commit to the database in batches or after each paper, and perhaps backup the database periodically (especially if running a long ingestion).

At the end of extraction and loading, we should have the full patient dataset in our database/graph, mirroring the manual table but potentially with more standardized coding and possibly additional info

gleaned from text. The **manual table provided can be used to validate** a subset of our automated output – for quality assurance, one could compare a few random entries.

7. Setting Up the Systematic Review and Query Interface

With data in place, we configure the interface for Track B and C queries. This might be a simple command-line or a web UI: - For a quick setup, a Jupyter notebook or Streamlit app could allow the user to enter a question which then triggers the orchestrator agent to run the systematic review steps. Alternatively, a small Flask web service with an endpoint `/query` that accepts a question and returns the answer JSON/HTML. - Implement the Query Parser Agent as a function or LLM call. Perhaps fine-tune a smaller model to classify query types (Is it asking about phenotypes frequency? gene-specific? treatment outcome? general summary?). Or use prompt engineering: e.g., a system prompt: *"You are a query analyst. Identify if the query is (A) data retrieval from patient DB, (B) requires new literature search, (C) pertains to patents/trials."*. Since our system has finite data, most questions should be answerable from it or via patent/trial fetch. - Based on the parsed intent, either trigger the systematic review pipeline or the patent pipeline. In code, this could be an if/else or using the AutoGen framework to route it. For Track B, the pipeline is: 1. Construct a database query (SQL or Cypher) to fetch relevant patient records. For example, if query asks "How many patients presented before 1 year of age?", we translate that to `SELECT count(*) FROM patients WHERE age_onset_months < 12`. If it's "Compare features in SURF1 vs MT-ATP6 cases", we do two queries or one grouping query. 2. Do the query and get results. 3. Feed results to LLM to summarize. Possibly provide the LLM with some stats or the actual list of patient data (if small) to derive a narrative. 4. Also, count references involved and have it prepare a PRISMA statement. We know total in DB = e.g. 248 patients (from 705 references). If our query yielded N patients from M references, we say that in PRISMA terms. 5. The LLM then generates text with citations. We can have it cite by [PMID] or [Reference number] that we provide. Because our knowledge graph knows which references each included patient came from, we can easily compile a list of included PMIDs. The agent could output something like (Smith et al 2020) style or just list PMIDs. For consistency with our UI here, maybe use our citation style or a simplified one. 6. Return the answer to the user.

- For Track C, integrate with external APIs:
    - Patent search could use an API like Lens or the European Patent Office's API. If unavailable, using an open dataset or a simplified approach: a prepared list of known patents relevant to Leigh (not ideal, better to actually query by keywords).
    - For ClinicalTrials.gov, there's an API or we can use their CSV export. But simplest: use the `requests` to query their API: e.g., `https://clinicaltrials.gov/api/query/study_fields?expr=Leigh+syndrome&fields=NCTId,Title,Status&min_rnk=1&max_rnk=1000&fmt=json`.
    - Parse JSON results and filter if needed (maybe filter out terminated trials, etc. depending on question).
    - Summarize with LLM as described earlier, then output to user.
- We should test a few example queries to refine the prompts and ensure the answers make sense and cite data. Because LLMs sometimes hallucinate, we rely on the structured data to ground answers. For instance, we might format the prompt to the Report Writer as:

```
Here is data for 50 patients (in CSV): ...
Question: "What are the common features in SURF1-related Leigh syndrome?"
Using the data above, answer the question with statistics. Cite the sources
by PMID.
```

This way, the model has actual data to base the answer on, reducing fabrication.

8. Running the System and Final Touches

At this stage, the system should be functional. Some additional steps for deployment: - **Background Scheduler:** It might be useful to have a scheduler (like a cron job or a Python schedule) to periodically check for new PubMed articles (e.g., monthly auto-update). This would involve re-running the metadata search, finding new PMIDs not seen before, and processing them (Track A pipeline) to update the database. This ensures the knowledge base stays current. - **User Interface:** Depending on the end-user (clinicians vs developers), a more user-friendly interface might be needed. For a clinician, a simple web interface where they can input queries or view the patient table is ideal. One could use a lightweight dashboard (Streamlit or Dash) to display the master table with filters (so one can browse patients by gene, etc.), and a query box for natural language questions. - **Documentation and Logging:** Ensure to document how each component is run. For example, how to start/stop the Docker containers, how to re-run ingestion for a specific paper, etc. Implement logging in the Python scripts (using `logging` module) so that if something goes wrong (like a PDF fails parsing, or an agent times out), we have a record of it. - **Testing:** Do a dry-run on a subset (like 10 papers) to validate the pipeline, then scale up. During the first full run, monitor resource usage. GROBID can use a lot of CPU when parsing many PDFs in parallel – consider processing sequentially or in limited batches to avoid overload. Similarly, LLM extraction can be memory heavy, so one at a time might be needed if using a large model on one GPU. - **Security:** If deploying as a service, ensure proper firewall or auth (especially if the interface is exposed on the internet). The data is derived from published papers (which are public), so there's no patient privacy issue per se, but one should still protect the system from unauthorized modification.

Once deployed, the user can trust that asking a question or running an analysis uses the most up-to-date literature. The deployment steps above can be scripted as well (for example, using a bash script or Makefile that sets everything up, or Docker Compose to run multiple containers). Packaging the entire system in a few Docker containers (one for the database, one for the app with agents, one for supporting services) is a good idea for portability.

Autonomous Agents: Roles and Prompt Designs

This system uses multiple specialized agents, each with a distinct role. Here we outline each agent's purpose along with the strategy for their prompting (what we "tell" the agent to make it function). The prompt engineering is crucial to guide the LLM behind the agent to produce the desired outputs.

- **Master Orchestrator Agent:** *Role:* Receives user input or high-level commands and determines which track or agent should handle it. It is essentially the manager.
  Prompt blueprint: A system prompt like: *"You are the Master Orchestrator, an expert project manager for a biomedical AI. Your job is to interpret the user's request and break it into tasks for your team of agents (Metadata, Extractor, etc.). Respond with a plan or by directly invoking the relevant agent."*

Behavior: This agent might not produce user-facing text often; instead, it might output a reasoning trace or a plan (if we implement it to explain decisions). It could say internally: "User asked for X, I will use Track B". Then it triggers sub-agents accordingly (in AutoGen, this could be done by sending the user query as input to another agent's conversation). The Orchestrator's prompt ensures it doesn't try to solve things itself but delegates. We might also provide it a list of agent names and capabilities as part of prompt so it knows who can do what.

- **Metadata Triage Agent:** *Role:* Assess new or existing article metadata to decide relevance and priority.
  Prompt: "You are a Literature Triage Agent. Given a list of article titles/abstracts and metadata, classify each as (Relevant case report / Irrelevant) and prioritize the processing order. Consider that relevant articles mention Leigh syndrome cases, and exclude pure reviews or animal studies. Output a list of PMIDs with labels 'include/exclude' and a brief reason."
  Example: For an input abstract, it might output: `PMID 12345: include (case report of 2 Leigh patients); PMID 12346: exclude (review article)`.
  Behavior: The Orchestrator would feed this agent with any new search results. The agent's output is then used to decide which PDFs to fetch and parse. It ensures we don't waste time on irrelevant papers.

- **Concept Extractor Agent:** *Role:* Extract raw information from text for each patient.
  Prompt: "You are an expert clinical information extractor. You will be given a case report section and you will extract key patient details. Focus only on facts explicitly stated. For each patient, output a JSON with fields: {patient_id, sex, age_at_onset, current_age_or_age_at_death, genetic_diagnosis, phenotypes_list, lab_findings, imaging_findings, treatments, outcome}. Use 'null' or [] if not mentioned. Use numerical values where possible (months for age <1 year, years for older)."
  We may include an example in the prompt for format, e.g., *"Example input: [some text]. Example output: { ... }"* to guide formatting.
  Behavior: The agent reads possibly large text (so ensure the model's context can handle it – might need 8k or 16k token model for long papers). It then yields structured JSON or a similar format. This is fed into our pipeline. If the text contains multiple patients, the agent either handles them one by one (we might split text and call agent multiple times per section), or we instruct it to output a list of JSONs for each patient in the input. Testing will show which is more effective. The key is making the prompt very clear on the expected output schema, to minimize the need for heavy post-processing.

- **Table Mapper Agent:** *Role:* Align extracted data to the master table format and fill any gaps by cross-checking the text.
  Prompt: "You are a Table Mapping Agent. You take JSON data extracted for a patient and map it to a fixed table schema of 294 columns. I will provide the JSON and the schema definitions. You output a JSON with all schema fields populated. If data is missing, leave it null. If something needs conversion, do it (e.g., convert '5 days' to 0.17 months for age fields). Also ensure each symptom in phenotypes_list is set in the respective column (e.g., HP:0001252 hypotonia -> fill 'hypotonia' column=1). Schema: [list of columns]."
  Given the schema is huge, we might not list all 294 in prompt, but perhaps instruct it to focus on categories. Alternatively, we handle mapping with code, not LLM, since 294 fields might be too much for the model to output reliably. Another approach: break mapping by sections (development, labs, neuro symptoms, etc.) and have the agent handle sections at a time to avoid confusion.
  Behavior: If using LLM, this agent will essentially transform one JSON structure to another, while adding default values (like if the table expects 0/1 for absence/presence of symptom, and the

extractor only listed present ones, then for each symptom not listed, fill 0). The agent should use the ontology: it can detect "ataxia" in phenotypes and knows HP:0001251 corresponds, and thus set ataxia column. We might supply it a condensed map in the prompt. E.g., provide a dictionary of symptom to column name or HPO to column. Or, more practically, do this mapping in Python, because it's deterministic rule-based (the Validator agent can then double-check the mapping if needed).

- **Validator Agent:** *Role:* Quality control on the structured data.
  Prompt: "You are a Validator Agent. You will be given the extracted structured data for a patient (in JSON or CSV form). Your task is to identify any anomalies, inconsistencies, or likely errors. Check for logical consistency (e.g., alive vs age of death, onset age not after current age, etc.), completeness (fields that are unusually empty given context), and compliance with ontology (e.g., if a phenotype text couldn't be mapped). List any issues found, or say 'OK' if everything looks consistent."
  Behavior: This agent works after the patient row is compiled. It can also have access to the original text snippet for reference if needed. For example, if the table has no phenotypes listed but the text clearly mentioned some, it could flag "No phenotypes recorded but text suggests symptoms were described." It might also catch formatting issues (like mutation not following expected format). The output is a list of issues per patient. The Orchestrator can decide what to do: potentially it could attempt a second pass to fix some issues (like if missing mapping, try a different mapping approach), or more likely, just log them for user review. The agent ensures higher confidence in the final data.

- **STORM Report Writer Agent:** *Role:* Generate human-friendly reports (narratives, summaries, even draft papers) using the data.
  Prompt: "You are STORM, an AI writing assistant specialized in medical reports. Your goal is to turn structured data or query results into a well-written report. Maintain objectivity and cite sources. When provided with a query and evidence data, produce a concise yet comprehensive answer, including statistics or patient counts as needed. Use a formal academic tone for reports. Embed reference identifiers (like PMID or [#]) to support factual statements."
  We might have different modes: *Summary Answer* vs *Full Report*. If a user just asks a question, a few paragraph answer with citations suffices. If a user requests a full "manuscript", we might instruct the agent to produce sections (Intro, Methods, etc.).
  Behavior: The agent will typically be invoked after data retrieval (Track B or C) to do the final composition. We provide it the relevant info in prompt. For example, for a systematic review query, we might give it an outline: "Background: X. Results: (we list some findings or even raw stats). Conclusion: ..." and let it fill in. Or just give it bullet points of findings and ask it to weave them into prose. It will reference publications via the citation style we decide (maybe superscript numbers or inline [PMID]). We must be cautious: the agent might hallucinate citations if it doesn't have them, so we feed it the actual source IDs. Possibly, we pre-format the evidence as "(Smith 2017)" in the content we give it, so it just incorporates those references naturally.

- **Digital Twin Composer Agent:** *Role:* Use the database to create simulated patient profiles or run hypothetical scenarios.
  Prompt: "You are a Digital Twin Composer. You create a comprehensive profile (digital twin) of a Leigh syndrome patient given certain parameters, using aggregated data. You can also simulate disease progression or outcome based on past cases. Provide a report of the simulated patient's life trajectory or outcomes of interest."
  Behavior: In current use, this agent might be invoked when a user says something like, "Create a

digital twin for a female patient with SURF1 mutation who starts symptoms at 6 months, and predict their likely clinical course." The agent would then gather from the database similar cases (female, SURF1, onset ~6m) – possibly the Orchestrator does a query and feeds the agent those cases' data. The agent then produces a composite narrative: e.g., "This digital twin, based on 20 similar cases, would likely develop hypotonia in infancy, with neuroregression by age 1 [5]. Most SURF1 cases (80%) required feeding support by age 2 [5], and mortality in this subgroup was ~60% by age 5 [5]. Our simulated patient might follow a similar course." This is forward-looking and not guaranteed, so the agent must phrase it carefully (perhaps we emphasize it to use probabilistic language and to note it's a prediction). This agent is more experimental, but as data grows, it could even incorporate a predictive model (for instance, a small machine learning model trained on the dataset to predict survival, which the agent could call as a tool).

For each agent, we will utilize the frameworks to set system prompts and roles. For example, AutoGen allows defining something like:

```
orchestrator = Agent("system prompt for orchestrator")
metadata_agent = Agent("system prompt for metadata triage")
...
autogen_session = Session(agents=[orchestrator, metadata_agent, ...])
```

Then the Orchestrator's logic might call `metadata_agent.run(input)` under the hood as needed. In Agent Zero, Agent 0 would literally issue instructions to sub-agents that are separate processes or threads. We might not implement a full back-and-forth conversation for all (some can be single-turn tasks, which is simpler).

Prompt Iteration: We anticipate refining these prompts. For instance, initial runs of the Concept Extractor might miss certain fields, so we adjust the prompt to highlight those. Or the Report Writer might produce too verbose output, so we might instruct it to be more concise. Since we have the advantage of controlling the models, we can iterate quickly.

Autonomy vs Human-in-the-loop: The agents are set to run autonomously, but we can include a mechanism where if the Orchestrator is unsure (low confidence scenario), it asks the user. E.g., "I found a term 'encephalopathy of Leigh?' which I'm not sure how to code. Should I treat it as Leigh syndrome?" But ideally, we minimize such interruptions by equipping the agents with enough knowledge and fallback rules.

In conclusion, each agent has a clear role akin to members of a research team. By carefully crafting their system prompts and using consistent output formats, the multi-agent system can achieve complex multi-step results in a controlled, verifiable manner. The modular design means we can update one agent's logic without affecting others (for example, if a better extraction model comes out, we can swap just the Concept Extractor's backend). The prompts and roles described will serve as the starting point for implementation; real-world testing will further tune them to perfection.

Developer & Coding-Agent Manual

This section is aimed at developers (human or AI coding assistants) who will implement and maintain the system. It breaks down the project into modules and tasks, providing a scaffold that can be followed step-by-step. We present it in a style akin to GitHub issues or a to-do list for clarity. Each task can potentially be handed off to a coding agent (like GitHub Copilot or GPT-4 in "developer mode") to expedite development.

Project Structure

Consider organizing the repository as follows:

```
leighextract/
├── data/              # input data (CSVs, PDFs)
├── outputs/           # outputs (extracted JSON, logs, etc.)
├── agents/            # prompts or classes for each agent
├── ingest/            # code for Track A (ingestion pipeline)
├── review/            # code for Track B (query and review)
├── patents/           # code for Track C (patent/trial search)
├── deploy/            # deployment scripts (Docker, etc.)
└── notebooks/         # Jupyter notebooks for prototyping (optional)
```

This structure helps separate concerns. Now, onto the tasks:

Task 1: Database Schema Design (Issue #1)
Description: Define the SQL schema for the patient data table and related tables. Also design the Neo4j graph model.
- [ ] Design `patients` table with columns matching the manual table (simplify if needed, e.g., use separate tables for labs or phenotypes if 294 columns too unwieldy). Consider using foreign keys for publication and gene to avoid redundancy.
- [ ] Define `publications` table (pmid, title, journal, year, etc.), `genes` table (gene symbol, perhaps full name).
- [ ] If splitting phenotypes: maybe a `patient_phenotypes` join table linking patient ID to HPO code (this might be easier than having hundreds of phenotype columns filled with 0/1). The manual table approach is denormalized; a normalized schema might be preferable for a live system. Decide and implement accordingly.
- [ ] For Neo4j, outline node labels: `Patient`, `Publication`, `Phenotype`, `Gene`, `Disease`; outline relations: `(:Patient)-[:REPORTED_IN]->(:Publication)`, `(:Patient)-[:HAS_PHENOTYPE]->(:Phenotype)`, `(:Patient)-[:HAS_GENE]->(:Gene)`, `(:Patient)-[:DIAGNOSIS]->(:Disease)`.
- [ ] Implement SQL DDL (either via SQLAlchemy models or raw SQL) and run it to create tables in Postgres. For Neo4j, maybe you won't create anything until insertion time (Neo4j is schema-optional, though you can constrain property existence if desired).

Task 2: PubMed Fetch Module (Issue #2)
Description: Write a script to fetch article metadata from PubMed and store in the database.

- [ ] Use Entrez or another API to execute the query and retrieve PMIDs. Code the function `search_pubmed(query) -> List[PMID]`.
- [ ] Code `fetch_pubmed_details(pmids) -> List[Record]` that returns parsed info (title, abstract, types, etc.). Use Biopython or parse XML.
- [ ] Insert these into `publications` table. (Upsert logic: avoid duplicates if running again). Also, output to a CSV for record if needed.
- [ ] If available, capture PMCID for each (look at the Record's fields or a separate ELink query to PMC).
- [ ] Implement any filtering in this stage (optionally flag which ones are case reports via PublicationType containing "Case Report"). Perhaps add a column `include_flag` initially set based on this.
- [ ] Test this on a small query (e.g., limit to 10 results) to verify data is inserted correctly.

Task 3: PDF Download Module (Issue #3)
Description: Create a downloader for article PDFs (and supplements).
- [ ] For each publication marked for inclusion, determine the download link. If PMCID is present, try using NCBI's utilities. Possibly implement a helper: `download_pmc_pdf(pmcid)`. For example, use `requests.get(f"https://www.ncbi.nlm.nih.gov/pmc/articles/{pmcid}/pdf/")` – often PMC provides a default PDF if available. If that doesn't work, parse the PMC HTML for a PDF link.
- [ ] If PMCID not available but DOI is, try using `doi2pdf` via crossref or Unpaywall. This could be tricky. Alternatively, skip those or require manual provision. Document that.
- [ ] Save PDFs into `data/pdfs/` directory, naming by PMID or PMCID.
- [ ] If supplements are needed: detect in PMC XML or HTML – they might be listed as "Supplementary Material". Possibly skip unless they obviously contain data we need (maybe not critical for initial run).
- [ ] Add error handling: if a PDF download fails, log it and continue. After running, the log can be checked and those PDFs obtained manually if crucial.
- [ ] Test on one known open access PMID (maybe try the examples from the CSV that have a PMC link).

Task 4: GROBID Processing (Issue #4)
Description: Write a function to process a PDF through GROBID and extract text.
- [ ] Ensure GROBID service is running (maybe implement a quick check or start it via subprocess if not using Docker – but Docker is simpler to manage outside code).
- [ ] Implement `process_pdf_with_grobid(pdf_path) -> tei_xml` (string or XML object). Use `requests` as shown in the guide [12].
- [ ] Parse the TEI XML to a Python object. Possibly create a dataclass or dict structure: e.g., `{'sections': [{'title': ..., 'text': ...}, ...]}`. We can use `xml.etree.ElementTree` to get all `<div>` sections and their content.
- [ ] Strip out references and figure legends from text if present (GROBID might include figure captions in body). It might also separate references section which we don't need now.
- [ ] Return or save the parsed text structure. We might store it in a JSON for intermediate inspection. Also keep original TEI if debugging layout is needed.
- [ ] Time performance: GROBID is quite fast (~2-5 seconds per PDF). If doing hundreds, it's fine but maybe not saturate it – could do sequentially or limited concurrency. If needed, implement simple multi-threading with a queue of PDFs (but careful with GROBID's own concurrency limits, default is usually fine).

Task 5: Table Extraction with Camelot (Issue #5)
Description: Integrate Camelot to extract tables from PDFs.
- [ ] For each PDF (or for those known to have tables, but easier to just attempt all), run Camelot. This could be done on the same loop as GROBID or separate after. Camelot might occasionally fail on certain pages;

wrap in try/except.
- [ ] If tables are found (`tables.n > 0`), inspect them. Perhaps we decide to only keep tables above a certain size (rows/columns) that likely contain multiple patients. Single-row tables might be trivial or not needed.
- [ ] Convert tables to Python data (Camelot's `table.df` gives a DataFrame). Save them either in a list or directly to file (e.g., CSV or JSON).
- [ ] Ideally, associate table with its parent article. Could store in database as well (like a `tables` table with a foreign key to publication and maybe table caption or something). But to keep it simple, may not store, just use on the fly in extraction.
- [ ] Note: Camelot cannot handle scanned PDFs – but since we focus on text PDFs, okay. If a particular PDF is image-only (rare for recent journals, but older ones maybe), Camelot returns nothing; we might then log "needs OCR" and skip for now.
- [ ] Test on one or two PDFs known to have tables (maybe pick a case series article). Confirm that the table content is correctly read (no obvious merging errors).

Task 6: Implementing the Extraction Agent Logic (Issue #6)
Description: Code the logic that uses text (and table) to produce structured patient data.
- [ ] Decide on approach: direct LLM vs rule-based or hybrid. The plan is to use LLM via Ollama. So, set up a function `extract_patient_data(text_sections, tables, paper_metadata) -> patient_dicts`.
- [ ] Within this function, handle splitting by patient. You might attempt simple rules first: for example, if `tables` is not empty and looks like it contains multiple patients, iterate its rows. Each row might correspond to a patient. In such case, you can map columns directly to fields. This is an easy win for case series where table summarizing patients exists. Then you might not even need LLM for those fields (just map table columns to our schema where possible). If table lacks detail (like narrative parts or outcome), supplement with text.
- [ ] If no useful table or for narrative parts like case reports: use the LLM. Construct a prompt as previously described. Possibly include the patient text. If multiple patients in text, split by an indicator. If no clear indicator, feed the whole and ask the model to output a list of patients. (Be careful: if the model confusion might be high with multiple patients at once. Another approach: split by paragraph and see if each starts a new case; many case series have each case in a separate paragraph or with headings.)
- [ ] Connect to Ollama: Use Python requests to call Ollama's local server (`http://localhost:11434/generate`) with the model and prompt. Alternatively, use `subprocess.run(["ollama", "run", model, "--prompt", prompt])`. The response will be a text that hopefully is JSON. Use `json.loads` to parse. Because LLM might produce non-perfect JSON, consider using a tolerant parser or regex to find the JSON snippet. Or instruct the model strongly: *"Output only valid JSON."*
- [ ] If the model output is incomplete or had errors, consider a retry or fallback. Possibly we could have a second prompt: *"Correct the JSON format"* if needed. Or do some minimal repair with Python.
- [ ] After getting the data dict, apply the ontology mapping functions (from HPO names to codes, etc.) as discussed. This might be outside the LLM to ensure consistency.
- [ ] The output of this function is likely a list of patient dictionaries (since one paper can have multi patients). Each dictionary has all fields ready to insert. Possibly also attach the source PMID for reference.

    - [ ] **Unit test** the extraction on one known case: perhaps take a paragraph from a Leigh case report (maybe from the CSV abstract or a known text) and see if the LLM returns a sensible JSON. Adjust prompt until it does.

Task 7: Data Insertion & Graph Update (Issue #7)
Description: Take the extracted data and insert into Postgres and Neo4j.
- [ ] Using an ORM or direct SQL, insert the patient row. If using an ORM (SQLAlchemy), define the model earlier or just use raw SQL prepared statements. E.g.,

```
INSERT INTO patients(pmid, patient_num, sex, birth_year, onset_age, gene, ... )
VALUES(...);
```

We should retrieve the new primary key (if any) or construct a patient_id like `PMID_patientNum` as used in the graph.
- [ ] Insert related entries: if a gene is not in `genes` table yet, insert it (or maintain a set to skip duplicates). If phenotype columns are separate, handle accordingly – but if we went with join table approach, insert rows into `patient_phenotypes (patient_id, hpo_id)`. This may be easier for an arbitrary number of phenotypes per patient.
- [ ] For Neo4j, use the Python neo4j driver to run Cypher queries as described. Ensure to use transactions and possibly parameterized queries to avoid Cypher injection if any. You can prepare a Cypher query string with parameters for one patient and run it. Or use the neo4j-embedded `MERGE` logic for each property as above.
- [ ] If using Weaviate, for each patient create an object vector. But generating a vector from structured data might not be straightforward – perhaps instead store their case description text in Weaviate for semantic search. Could take the compiled narrative (like a concatenation of all info) and use Weaviate's text2vec (if using their transformer) or manually compute embedding if we have model. This could be an enhancement, not critical for initial deployment.
- [ ] Commit transactions. Possibly wrap the whole ingestion of one paper's patients in a transaction so either all or none of that paper's data is inserted (to avoid partial if something fails mid-way).
- [ ] Log completion: e.g., print or log "PMID X done: Y patients extracted". This helps track progress.

Task 8: Query Handling (Issue #8)
Description: Implement the query system for systematic review (Track B).
- [ ] Write a function `answer_query(user_query)`. This will be called by a UI or CLI. It should: - Determine query type: Perhaps maintain some simple rules (if query contains "trial" or "patent" -> Track C, else Track B by default, but confirm with LLM if ambiguous). We might actually run it through an LLM classifier (small one) or just route to orchestrator agent and let it figure out. For coding simplicity, start with if/else keywords.
- If Track B: * If query is something like "How many...", "What proportion...", etc., it implies a quantitative query. We can attempt to parse it to an SQL automatically. There are libraries or can do a small custom mapping. Alternatively, skip parsing and directly feed to an LLM that has access to the data (this is harder if data is large; better to do structured). * A compromise: for known types of questions, prepare some templates. Example: "How many patients have X?" -> `SELECT COUNT(*) FROM patients WHERE X_column=1`. "What is the average age of onset for Y?" -> `SELECT AVG(onset_age) FROM patients WHERE condition`. This requires some NLP. Possibly use an agent (Query Parser) that outputs a pseudo-SQL which we then execute (safer than letting it run actual SQL on its own). * Implement a few sample question parses and refine as needed. If a question cannot be parsed easily, just retrieve all patient data (or relevant subset) and let the LLM do it. For instance, for "Compare feature A between group1 and group2", we might retrieve two subsets and then prompt LLM to compare. * Retrieve data: for efficiency, we could use SQL for numeric answers or small subsets. If the subset is large (like "summarize all cases"), we might

not want to dump all ~250 patient records into LLM context at once. But maybe we can compress – e.g., compute key stats via SQL and just feed those. * Use the Report Writer agent with a prompt containing either the summary stats or a truncated list of results. * For citations: if the summary is derived from all data, citing multiple sources is tricky. We might just cite a few representative references (like the earliest or a review) or simply say "data from X cases across Y studies". Alternatively, we could attach a reference list at end. - If Track C: * Call a `search_patents_trials(query)` function that uses APIs as described. Get results lists. Possibly shorten them to top 5 each. * Format a summary (maybe directly ask the LLM: "Summarize these patents and trials info:" and give it a structured input of titles, etc.). * Return the output. These likely will cite the patent numbers or NCT IDs, which is fine (we can treat those as citations in context). - [ ] Ensure the function returns a nicely formatted Markdown string (because our output here is Markdown). So include any lists or bold text as desired. The Report agent might already produce Markdown if instructed (or we can do minor formatting to its text). - [ ] Testing: simulate a few queries in code and print the outputs.

Task 9: Agent Orchestration Integration (Issue #9)
Description: Tie the specialized functions into the multi-agent framework (AutoGen or custom).
- [ ] If using AutoGen, define agents like:

```
OrchestratorAgent = autogen.Agent(prompt="... from earlier ...")
MetadataAgent = autogen.Agent(prompt="...")
...
session = autogen.Session(agents=[OrchestratorAgent, MetadataAgent, ...])
```

AutoGen allows one agent to call others by name in the conversation. E.g., Orchestrator can output: `@MetadataAgent Please triage these references: ...` and then that agent sees that as input. We will incorporate in prompts guidelines like "If you need to, call an agent by using @AgentName and then your request." This is somewhat advanced; initial version might not do full conversation, instead we might orchestrate in Python logic (which is simpler to implement). E.g., write a procedural script for ingestion rather than letting agents spontaneously converse. That's fine for now. - [ ] For Agent Zero, if we were to use it, it would likely involve launching an Agent0 process. That might be overkill for initial deployment. We can mention in documentation that Agent Zero could be used in interactive mode if the developer wants to allow the system to self-improve or do more autonomous exploration (like writing its own code tools). - [ ] CrewAI, if any usage, might come in building a flow definition. Possibly skip actual integration due to time; instead maybe just comment that in future, one could use CrewAI's GUI or YAML flows to orchestrate tasks in a more declarative way. - [ ] Essentially, our code itself can serve as the orchestrator. The "agents" are implemented as functions or API calls to LLM with distinct prompts. So ensure these prompts are stored (maybe in `agents/prompts/`) and can be tweaked without altering code. - [ ] Provide a top-level script or notebook: `run_full_ingestion.py` that will call everything: fetch metadata, loop PDFs, run extraction, etc., using the functions above in sequence (this is Track A end-to-end). And a `ask_question.py` for Track B/C usage.

Task 10: Testing & Validation (Issue #10)
Description: Verify the system with known data and prepare it for user.
- [ ] Compare a subset of the extracted data against the manual table for consistency. For instance, pick 5 patients from our DB and find them in the manual Excel to see if key fields match. Document discrepancies and investigate (this can reveal if our extraction missed something or if the manual had an error). - [ ]

Performance test: time how long a full run takes for, say, 10 papers, and extrapolate. If very slow, consider optimizations (e.g., batching LLM calls if possible, or using smaller models). - [ ] Make sure all external dependencies are accounted for. If any require API keys (Entrez, possibly some patent APIs), list them clearly in documentation and possibly provide placeholders/config file. - [ ] Create example outputs: e.g., run a query "List the most common symptoms in Leigh syndrome" and capture the answer. Use this as a demonstration in documentation to ensure everything is working. - [ ] If possible, involve a clinician or domain expert to review a sample of outputs (especially the summary answers) to ensure medical accuracy.

Many of these tasks can be parallelized by a team or even delegated to coding agents one by one. For example, one can prompt a coding AI: *"Implement a Python function that given an HPO name returns the HPO ID, using a provided dictionary of synonyms."* This could expedite development. It's crucial to keep the agents (or developers) aware of the big picture: the goal is accurate data extraction. So writing unit tests for each piece (where feasible, e.g., test ontology mapping separately) will help maintain quality.

Throughout development, use version control (git) to track changes and possibly issues. If working with an AI pair-programmer, ensure to validate its outputs due to the critical nature of the data.

This manual should serve as a road-map. Checking off each issue will progressively build the system from scratch to a fully operational state. The modular approach ensures that if one component fails or needs improvement (say the LLM extraction), it can be debugged or upgraded in isolation.

To-Do List for the User / Maintainer

Once the system is up and running, the job isn't completely finished – maintaining and utilizing the system requires some ongoing tasks. Here's a checklist for the user or maintainer to ensure smooth operation and continuous improvement:

- **Obtain Access to New Data Sources:** Initially, we focus on PubMed/PMC open access literature. The user should arrange access to any paywalled articles that were skipped (if those contain important data). This might involve getting PDFs via institutional subscriptions or contacting authors. Once obtained, those PDFs can be added to the pipeline manually. Additionally, if there are patient registry data or other databases to integrate in future, gather those credentials or dumps (for example, a mitochondrial disease registry that could feed into the digital twin simulation).

- **Regularly Update Literature:** Schedule time (e.g., monthly or quarterly) to run an update. New Leigh syndrome cases might get published; running the metadata search again will catch them. The user should feed those into the system (the pipeline can be rerun in update mode). Also, check for updates in ontologies (HPO and others release updates periodically) – pulling the latest HPO definitions will keep the system current with new terms.

- **Review Flagged Extractions:** The Validator Agent will produce flags for potential issues. The user should review these flagged items by going back to the source text. For instance, if patient 42 was flagged "Gene name uncertain," the user can open that article and clarify (maybe the article had a typo or a novel gene). After review, update the database accordingly (the system might have a manual override interface, or one can directly edit the DB entry). Maintaining a **"gold-standard" set**

of a few fully verified patient entries is wise – these can be used to recalibrate the extraction agent (for example, fine-tuning an LLM or updating prompt examples).

- **Provide Feedback and Corrections:** If the user notices systematic errors (e.g., the agent consistently mis-identifies "failure to thrive"), they should update the ontology mapping or add prompt instructions to fix that. This might involve adding new synonyms to the HPO dictionary or tweaking the Concept Extractor's prompt with additional examples. Document these changes (perhaps in a CHANGELOG or in comments) so improvements are tracked.

- **Monitor Performance and Resources:** Keep an eye on the system's performance. If using a server, monitor CPU/RAM usage during large ingestion. The user might need to adjust some settings (like how many threads or which model sizes) to fit their hardware. Also monitor the database sizes – after adding hundreds of patients, queries might need indexing (e.g., index on gene or on phenotypes table). Add indexes as needed to keep query speed up.

- **Curate and Merge Duplicates:** Because the data comes from literature, some patients might appear in multiple papers (e.g., a patient in an initial case report might reappear in a follow-up or in a review). The system might treat them as separate entries. The user should identify if any entries likely refer to the same individual (perhaps via unique mutations + institution clues). If so, consider merging their data or at least linking them in the graph ("possibly same patient as…"). This is more of a curation task and may be very rare, but it's a possibility in literature-derived data.

- **Extend Ontology Coverage:** If the user finds that certain extracted terms aren't getting mapped (e.g., a lab test not in our LOINC list, or a symptom that isn't in HPO), they should extend the system's ontology mappings. This could mean updating the dictionary or even contributing a new term to HPO if it's truly missing (the HPO project allows term requests). For example, if "3-methylglutaconate" wasn't mapped, ensure we add the proper code or at least mark it as a known gap.

- **Utilize Digital Twin & Simulation Carefully:** When using the digital twin composer for predictions, treat the outputs as hypotheses. The user (likely a clinician) should review those simulated outcomes and not rely on them blindly. If something seems off (e.g., the simulation predicts an outcome not seen in actual patients), that might indicate either insufficient data or a quirk in the model. The user might then refine the simulation approach or just note that more data is needed. Provide those insights back into system development (maybe as future improvement tasks).

- **Document and Publish Findings:** The ultimate goal could be to publish a comprehensive review or database of Leigh syndrome. The user should use the system to generate such outputs (with the STORM writer) and then carefully edit them. The system can draft a lot of the writing (methods section describing how this was done, tables summarizing data, etc.), but the user must ensure the final publication meets scientific standards. Part of the to-do might be preparing figures (maybe an automatically generated figure of phenotype frequencies – the user can use the data to make plots). The system eases the data gathering, but the human expertise is still required for interpretation and final presentation.

- **Manage Credentials and APIs:** Keep track of any API keys (NCBI, etc.) used. If they expire or quotas are hit, the user should update or manage usage. For instance, NCBI now requires API keys for

heavy use – ensure the key is kept private but available to the system in configuration. Similarly, if using any paid service for patents (Lens has limits), monitor usage or use a local dataset.

- **User Training and Handover:** If the user is an individual who built this and intends to share with a team, ensure there's a proper readme (which this document forms the basis of) and perhaps a training session to show others how to run queries or add data. Because the system is autonomous, you want users to trust but verify it – encourage them to inspect sources via the linked PMIDs if they have doubts about an answer.

By regularly performing these tasks, the user will keep the system **accurate, up-to-date, and useful**. Think of the system as a living project: the more it's fed with feedback and new data, the smarter and more valuable it becomes. Conversely, neglecting it (e.g., not reviewing flags or not updating with new papers) could let errors accumulate or data become outdated over time.

Future Expansion Plans

Looking ahead, there are several exciting directions to expand this system beyond its current capabilities, turning it into a comprehensive platform for research and clinical insights in Leigh syndrome and potentially other disorders:

- **Digital Clone Simulation:** Building on the Digital Twin concept, we can create virtual patient "clones" that simulate the progression of disease under various conditions. This could involve integrating computational models of mitochondrial disease. For example, one could plug in a pathophysiological model that uses the patient's genotype and phenotype to simulate energy production deficits over time, thereby predicting disease trajectories. The system could allow one to tweak parameters (say, simulate if a patient had a higher residual enzyme activity) and see the outcome. This would be invaluable for hypothesis testing – e.g., what if a certain therapy improved complex I activity by 20%? The digital clone might simulate a milder course, which can guide real-world therapeutic ideas. Achieving this will require collaboration with biomedical modelers and possibly integrating with simulation tools or languages like R or MATLAB for advanced modeling. The agent architecture is already suited to call external tools, so an agent could be designated to run such simulations and report back.

- **Natural History Analytics:** With an enriched longitudinal dataset (especially if we incorporate follow-up info from multiple papers), the system can perform natural history studies. This means quantitatively analyzing how the disease progresses on average – time from onset to loss of ambulation, survival curves, etc. We could implement statistical analysis modules: for instance, use Kaplan-Meier analysis on age of death or time to wheelchair dependence. A future version could have an *Analytics Agent* that, upon request, computes these and maybe even generates plots. As more data accumulates, these analytics become more robust. Additionally, we might integrate other data sources like patient registries or genetic databases to augment the literature data, giving a fuller picture of natural history.

- **Virtual Clinical Trials:** With the patient database and simulation capabilities, we can explore in-silico clinical trials. For example, select a cohort of "virtual patients" from our data (or simulate new ones matching certain criteria) and then apply a "virtual intervention" (like assume a drug that reduces lactate by X% or a gene therapy that corrects a mutation) and run the simulation forward to predict

outcomes vs. a control cohort. The system could use Monte Carlo methods to account for variability. Results might indicate whether a trial is likely to show a benefit, what sample size might be needed, etc. This would not replace real trials, but could help in designing them. We could also incorporate trial data (from Track C) to validate these simulations – if a trial reported some outcomes, compare with our predictions.

- **Automated Publication (STORM integration):** We plan to leverage **STORM** – Stanford's open-source tool for AI-generated content – more fully. This means the system could automatically write up findings as a draft publication or report. For example, once the data is updated, it could generate an updated "Leigh Syndrome Annual Report" with the latest stats, newly reported cases, and any changes in trends. Using STORM ensures the style and completeness of the write-up is high. We can set it to generate everything from an abstract to conclusions, possibly even format references properly. This could be extended to write patient case summaries on the fly for reports or generate educational materials (like an overview of Leigh syndrome for a medical audience) by repurposing the knowledge base.

- **Lens-like Unified Interface:** In the future, our system could serve as a **one-stop "Lens" for Leigh syndrome**, where literature, patents, clinical trials, and even *omics* data are all interconnected. This means developing a user interface where a user can toggle between viewing scientific publications, viewing relevant patents (with maybe full text search of patent documents using our LLM to summarize them), and viewing trial registrations. One could envision something like: search a gene -> see all patients in our database with that gene, see any therapy patents targeting that gene, and see trials involving that gene. Achieving this might involve indexing patents and trial descriptions in a similar way to how we did literature, and expanding the knowledge graph to include those connections (e.g., link a Gene node to Patent nodes if a patent is about that gene). This is quite feasible since a lot of the framework is similar (text ingestion and summarization). It would turn the system into a *knowledge hub* for the disease.

- **Multi-Disease Generalization:** While the current system is tailored to Leigh syndrome, the architecture can be extended to other conditions, especially in the mitochondrial disease spectrum or neurology. In the future, one could clone the pipeline for another disease (changing the search queries and ontologies accordingly). Eventually, maintain a multi-disease knowledge base. The agents could be made context-aware of disease (perhaps with a variable or by detecting from query) and then fetch from the relevant dataset. This would amplify the utility – effectively a framework for rare disease knowledge extraction.

- **Integration with Clinical Data (EHRs):** Another frontier is linking this literature-derived data with real-world clinical data. For instance, if a clinician has a Leigh syndrome patient, they could input the patient's profile to the system (through a Digital Twin agent) and the system could compare that profile with the closest matches in literature (using our vector search) and provide insights (like "Patient matches cases in literature X, Y – those patients had outcome Z, consider monitoring for …"). This would make it a clinical decision support tool. To do so, we'd need to ensure data privacy and possibly integrate with hospital systems (maybe via FHIR standard). The ontology mapping we did would facilitate this integration, since HPO and SNOMED are used in some EHR contexts.

- **Improved Autonomy and Learning:** With Agent Zero's framework, we can allow the system to continuously learn from user interactions and corrections. For example, if the user corrects an

extraction, the system could update its prompt or fine-tune an internal model to not repeat that mistake. We could also implement a *reward mechanism* for the agents – e.g., the Validator could signal success or failure of an extraction, which in Agent Zero's paradigm could allow the agent to adjust its strategy (it might try a different prompt or consult an external knowledge source next time). Over time, the agents become more accurate and require less human intervention. Essentially, turning it into a semi-autonomous researcher that improves with experience.

- **Collaborative Knowledge Graph (Community Curation):** We could allow multiple users (or experts worldwide) to contribute to the knowledge base. For instance, a clinician could input an unpublished case into the system (via a form) to enrich the data. Or experts could verify certain data points, which gets marked as "validated by expert". This moves towards a crowdsourced database, somewhat like ClinVar does for variants but here for case phenotypes. Technically, this means building a user management system and a front-end for data entry, which is beyond current scope but aligns with long-term goals of creating an open resource.

- **Applying to STORM/LENS Workflows:** The mention of STORM and LENS hints at integrating our work with existing tools or platforms. STORM might allow plugging our content generation into, say, a journal's submission system for rapid drafting. Lens.org might allow custom datasets – perhaps we could feed our database into Lens as a collection. Conversely, we can use Lens's APIs to automatically get patent full texts for deeper analysis with LLMs (embedding those into our system for summarization). The synergy with such platforms could raise the project's visibility and utility.

In summary, while the current system achieves automated data extraction and answering questions, these future plans aim to transform it into a **holistic AI-driven research assistant and simulation platform**. The advancements would push the boundaries from merely reading and summarizing research to actually predicting outcomes and generating new hypotheses. With ongoing development and collaboration, this system can significantly accelerate both research and clinical understanding of Leigh syndrome and similarly complex conditions, ultimately contributing to better patient outcomes through informed insights and hypothesis generation.

The roadmap above is ambitious, but each step builds on our existing foundation. By keeping the system modular and using ontologies and standards, we've ensured that expansions (whether breadth-wise to new data or depth-wise to new analytics) can be done without starting from scratch. The future is bright for such AI-driven biomedical intelligence systems, and this project is well-poised to be at the forefront of that evolution.

---

1 2 3 9 10 11 12 GitHub
https://github.com/mprestonsparks/DocGen/blob/0c59e9dbf1ba50396e7c780b057ba87284e43426/docs/paper_architect/components/paper_extraction/grobid_integration.md

4 The AI-Native, Open Source Vector Database - Weaviate
https://weaviate.io/platform

5 PRISMA statement
https://www.prisma-statement.org/

6 AutoGen - Microsoft Research
https://www.microsoft.com/en-us/research/project/autogen/

7 Run LLMs locally with Ollama on macOS for Developers - DEV Community
https://dev.to/danielbayerlein/run-llms-locally-with-ollama-on-macos-for-developers-5emb

8 Camelot: PDF Table Extraction for Humans — Camelot 1.0.0 documentation
https://camelot-py.readthedocs.io/en/master/