ChatGPT

Section A: Public Biomedical APIs for Integration

Below is a curated list of open-access biomedical APIs (free for research/commercial use) that can enrich the system. For each API, we outline what data it provides, how to access it, and key integration details:

ClinicalTrials.gov API (v2) – Clinical Studies Registry

- **Description:** Official U.S. registry/results database of clinical trials worldwide, providing structured study info (protocol, status, outcomes, etc.) [1]. The API returns trial records (study metadata, eligibility criteria, locations, results summaries, etc.).
- **Data Types:** Clinical trial metadata (titles, conditions, interventions, phase, status, sites, outcomes) and results data when available. Each trial is identified by an `NCT` ID.
- **Authentication & Access: No API key required.** The REST API is public and open; base URL: `https://clinicaltrials.gov/api/v2/` (OpenAPI spec available) [2]. No signup is needed [3].
- **Rate Limits:** ~50 requests/minute per IP address [4] (no daily cap noted). Applications should pause on HTTP 429 errors.
- **Pagination & Query Syntax:** Supports complex filters and is paginated by default (10 studies per page, up to 1,000) [5]. Query parameters include condition (`query.cond`), intervention (`query.intr`), location (`query.locn`), status filters, etc. Multiple search fields can be combined (AND semantics) [6] [7]. Use `fields` param to select specific fields and `countTotal=true` to get total count [8] [9].
- **Integration Notes:** The adapter can accept an NCT ID or search query. On fetch, it calls the `/studies` endpoint with appropriate `query.` parameters, then pages through results if needed. It should parse JSON fields into our IR (e.g., trial title, status, arms, outcome measures as `Document` metadata and individual fields as `Block` entries). The API returns JSON by default. No authentication needed, but implement retry/backoff for rate limit compliance.
- **Sample Request:** *Get melanoma trials involving BRAF inhibitors* – for example:

```
curl -X GET "https://clinicaltrials.gov/api/v2/studies?
query.cond=Melanoma&query.intr=BRAF"
```

This returns matching trial records in JSON [10]. Each record's sections (e.g. `protocolSection` with eligibility, `resultsSection` for outcomes) can be extracted and mapped to IR fields.

FDA openFDA API – Drug and Device Data

- **Description:** OpenFDA is an Elasticsearch-based API serving public FDA datasets [11]. It provides information on drugs (e.g. adverse event reports, product labels), medical devices (adverse events, recalls), and more. It's designed for easy search across FDA's publicly available data [12].
- **Data Types:**
- *Drug Adverse Events (FAERS):* Reports of drug side effects and medication errors.
- *Drug Labeling (SPL):* Structured product labels with indications, dosages, warnings, etc.
- *Device Events (MAUDE):* Medical device incident reports.

- (Also endpoints for food recalls, device recalls, etc.) Each endpoint has unique fields (e.g. drug event reports include patient demographics, reaction outcomes [13]).
- **Authentication & Access: API key optional.** The base endpoints are under `https://api.fda.gov/`. No login is required, but obtaining a free API key is recommended to increase limits [14] (get one via open.fda.gov). Without a key, you are limited to 40 requests/min; with an API key, 240 requests/min [15]. Burst limit ~4 requests/sec [15].
- **Rate Limits & Usage Policy:** ~1,000 requests/hour without key (40/min) and up to ~120k/day with a key [15]. The API is public domain; data is provided under an open license (with disclaimers in the `meta` field). Heavy users should cache results and respect the "please limit to 100k calls/day" guideline [16].
- **Query Format & Pagination:** Queries are RESTful with a common syntax. Use the `search` parameter for filtering, with field-specific queries (Lucene syntax). For example, to find adverse events involving aspirin: `search=patient.drug.medicinalproduct:aspirin`. Pagination is via `limit` (max 100 or 1000 depending on endpoint) and `skip` parameters [17]. By default, `limit=1` if not specified [18]. The JSON response contains `meta` (total count, license, last updated) and `results` (array of records) [19].
- **Integration Notes:** We will implement separate adapters for needed endpoints (e.g., an **OpenFDA AdverseEvent Adapter** and **OpenFDA Label Adapter**). Each adapter will construct the query URL with appropriate filters (for example, by drug name or device ID). The adapter should map JSON fields to IR: e.g. each FAERS report becomes a `Document` with `meta.patient_age`, `meta.drug_name`, etc., and adverse reactions listed as `Block` items. For drug labels, sections (Indications, Boxed Warnings) are already structured (SPL section IDs) and can map to IR sections. **Important:** Include the API key in requests if configured, and handle 429 responses by backing off.
- **Sample Request:** *Fetch one drug adverse event report* – e.g.:

```
curl "https://api.fda.gov/drug/event.json?limit=1"
```

This returns a single FAERS report in JSON [18]. Similarly, `drug/label.json?search=indications:melanoma&limit=10` would retrieve drug labels mentioning "melanoma". These can be tested interactively on open.fda.gov.

NIH RxNorm API – Drug Vocabulary Service

- **Description:** The RxNorm API (by NIH/NLM) provides programmatic access to the RxNorm dataset, a standardized nomenclature for clinical drugs [20]. It allows lookup of drug identifiers, names, and relationships (e.g. ingredient to brand mapping). RxNorm is **fully open-access** – no license or account needed [21] [22].
- **Data Types:** Standardized drug names and codes. Key data includes **RxCUI** (unique concept IDs for drugs), ingredient names, brand names, dose forms, National Drug Codes (NDCs), and relationships (ingredients → brand, etc.). The API supports functions like name normalization (find an RxCUI by drug name), listing all NDCs for a drug, getting properties and relationships of a drug concept [23] [24].
- **Authentication & Access: No auth required.** Base URL: `https://rxnav.nlm.nih.gov/REST/`. Endpoints are RESTful (returns XML or JSON if `.json` is added). Usage is open to the public under NLM terms (RxNorm data is U.S. government public domain) [21].

- **Rate Limits:** Not formally documented; generally robust for moderate use. (NCBI may apply default E-utilities limits ~3 requests/sec/IP if abused, though RxNorm API is separate.) It's advisable to throttle if doing large volumes and to use caching for repeated queries.
- **Integration Notes:** The adapter can use endpoints like:
- `GET /REST/rxcui?name={drugName}` to find the RxCUI for a drug name (normalized match) [25].
- `GET /REST/drugs.json?name={name}` for a list of related drug concepts (ingredient, brand, etc.) [26].
- `GET /REST/rxcui/{RxCUI}/allproperties.json` for all details of a concept.
The API returns structured data; e.g., `findRxcuiByString` gives the RxCUI and name for an input string. Integration will involve mapping these outputs to our knowledge graph – e.g., mapping drug mentions to RxCUI codes. The **RxNorm adapter** can be used in ETL jobs to normalize drug names found in text (e.g., from trial data or publications) to standardized identifiers.
- **Sample Request:** *Find the RxCUI for "aspirin"* – for example:

```
curl "https://rxnav.nlm.nih.gov/REST/rxcui.json?name=aspirin"
```

This returns the RxNorm concept ID and name(s) for aspirin. Similarly, `.../drugs.json?name=Cymbalta` returns all drug records matching "Cymbalta" (with ingredient and brand info) [26]. These JSON responses can be parsed to extract drug identifiers (to store in IR `meta` fields like `meta.rxnorm_id`). No special auth or keys are needed.

EMBL-EBI ChEMBL API – Bioactive Compound Database

- **Description:** ChEMBL is a large open database of bioactive molecules (drugs and drug-like compounds), including their chemical structures, bioassay results, and targets [27]. The ChEMBL API provides programmatic access to this data, enabling queries for compounds, targets, and activities.
- **Data Types:** Small-molecule data: compound structures (SMILES, InChI), molecular properties, mechanism-of-action, and bioactivity data (IC50, EC50, etc. from assays). Also includes target information (proteins, gene targets) and assay descriptions. Each compound has a **ChEMBL ID** (e.g. CHEMBL25 for aspirin). Bioactivities link compounds to targets with assay results.
- **Authentication & Access: No API key required for basic use**, but rate limiting is enforced. Base URL: `https://www.ebi.ac.uk/chembl/api/data/` (REST endpoints returning JSON or XML). The API is open-access; ChEMBL data is CC BY 3.0 licensed. **Rate Limit:** ~1 request/second for unregistered users [28]. For higher throughput, EBI offers an API key or encourages downloading the database. (Per a 2025 update, users without a key are limited to 1 rps; with a key or partner arrangement, higher rates can be negotiated [29].)
- **Pagination:** All list endpoints are paginated. Default page size is 20; use `limit` and `offset` params to page through results. Responses include `total_count` and links for `next` / `prev` with offsets [30]. For example, searching activities might return a JSON with `"next": ...?offset=20` for the next page.
- **Integration Notes:** The **ChEMBL adapter** can be used to enrich our knowledge graph with drug information. For instance, given a drug name or ChEMBL ID from another source, we can fetch compound properties or bioactivities. We will use the REST endpoints such as:
- `/chembl/api/data/molecule/{ChemblID}.json` – retrieve a compound's details (formula, properties, synonyms, etc.).
- `/chembl/api/data/mechanism/{ChemblID}` – get mechanism of action.

- `/chembl/api/data/indication/{ChemblID}` – clinical indications.
  In integration, for each compound we ingest, we map important fields to IR: e.g., `Document` for a compound with `meta.chembl_id`, and child `Block` entries for properties (molecular weight, targets, etc.). We will also leverage the official **Python client** (`chembl_webresource_client`) for convenience [31]. Long queries (like lists of many IDs) can use POST with the `X-HTTP-Method-Override: GET` header as per ChEMBL docs [32]. Retries and backoff will be important due to the rate limit.
- **Sample Request:** *Lookup a compound by ChEMBL ID* – e.g.:

```
curl "https://www.ebi.ac.uk/chembl/api/data/molecule/CHEMBL25.json"
```

  retrieves the molecule record for CHEMBL25 (which is Thalidomide in ChEMBL's test data). Or, to find by name, one can query the cheminformatics service:

```
curl "https://www.ebi.ac.uk/chembl/api/data/chembl_id_lookup/
{ChemblID}.json"
```

  (our adapter will use the lookup service to find the appropriate entity type for an ID [33]). The JSON result will be parsed and its fields like `molecule_structures`, `properties`, and `therapeutic_flags` will populate the IR fields.

HL7 FHIR APIs – Electronic Health Record Data (Synthetic)

- **Description:** *FHIR (Fast Healthcare Interoperability Resources)* is a standard for exchanging healthcare data (EHR records). While patient EHR data is protected, there are publicly available **FHIR sandbox APIs** with synthetic data (e.g., HAPI FHIR servers loaded with Synthea-generated patients). These allow testing integration with EHR-like resources (Patients, Observations, Conditions, etc.).
- **Data Types:** Patient records (with demographics, encounters, diagnoses, lab results, medications, etc.) represented as structured resources (JSON or XML). For example, **Patient** resource for demographics, **Condition** for diagnoses (with ICD-10/SNOMED codes), **MedicationRequest** for prescriptions, etc. Synthetic datasets like Synthea simulate real patient data for a region.
- **Authentication & Access:** Many FHIR test servers are open with no auth. For instance, the public HAPI FHIR test server at `https://hapi.fhir.org/baseR4` requires no API key. Some other sandboxes (like SMART HealthIT) require registering an app but are free. We will focus on open endpoints (no OAuth needed).
- **Usage & Rate Limits:** FHIR servers typically have generous read limits for testing (the HAPI server allows a few hundred requests/minute, but is shared community resource). Bulk data requires pagination (via `?_count` and `next` links). FHIR queries use standard REST: e.g. GET `[base]/Patient?name=Smith`. Responses contain resources in JSON with hyperlinks for paging.
- **Integration Notes:** To incorporate EHR-like data (though real PHI is out of scope [34]), we can use a FHIR adapter to ingest *synthetic patient data or public clinical datasets* for analysis. For example, we might ingest **MIMIC-IV demo data** if available via FHIR. The adapter can fetch resources by type: e.g., GET all Observations for a patient. We will map FHIR resources to our IR as follows: each resource can be a `Document` (with `meta.resourceType` and `meta.patient_id`), and key elements as `Block` entries (e.g., a Condition's code, description, onset date). We will leverage

FHIR's standard fields and ontologies (e.g., using ICD-10 codes, SNOMED CT, LOINC as provided in the data) to normalize medical concepts.

- **Sample Request:** *Query a FHIR server for a patient's data* – for example:

```
curl "https://hapi.fhir.org/baseR4/Observation?patient=12345&_count=10"
```

retrieves up to 10 Observations for patient with ID 12345 from the HAPI server. The JSON will include each Observation's details (code, value, timestamp, etc.). In our system, the FHIR adapter could use such queries to pull all relevant resources (patients, their conditions, meds, etc.) to build a synthetic patient profile in the knowledge graph. (Note: For production use with real EHRs, OAuth2 and stricter security would be required, but for open synthetic data we can access directly.)

World Health Organization ICD-11 API – Global Disease Ontology *(Optional)*

- **Description:** ICD-11 is the WHO's global diagnostic classification. The **ICD-API** allows programmatic lookup of ICD-11 codes, descriptions, and hierarchical relationships. This helps in normalizing disease terms to standard codes.
- **Data Types:** Disease and health condition codes (ICD-11) with metadata like definitions, synonyms, parent/child relationships. Useful for mapping extracted conditions (from text or trials) to a standard taxonomy.
- **Authentication & Access: API key required (free)** – users must request a Client ID/Secret from WHO and obtain a token via OAuth2 [35] . Once authorized, requests are made to endpoints like `https://id.who.int/icd/release/11/2023-05/mms` (for the ICD-11 release). The API is free for use; ICD-11 content is under a Creative Commons license.
- **Rate Limits:** Not publicly specified, but practical use suggests moderate limits (e.g. a few thousand calls per day). The API is primarily for lookups, not bulk download (WHO provides the entire ICD-11 in a release file for bulk use).
- **Integration Notes:** We can integrate ICD-11 by adding a terminology adapter. For example, when ingesting trial eligibility criteria or guideline data, our system can call the ICD API to map condition names to ICD-11 codes for the knowledge graph. The adapter will use the token-based auth to call endpoints like `GET /icd/release/11/{code}` or search by text. The returned JSON provides the title (diagnosis name), code, and lineage. We will store these codes (or attach them to condition entities in our graph).
- **Sample Request:** *Find an ICD-11 code by its title* – e.g., using the ICD API's search:

```
curl -H "Authorization: Bearer <token>" "https://id.who.int/icd/release/
11/2023-05/mms/search?q=diabetes"
```

would return matching ICD-11 entities for "diabetes". Each result includes the code (e.g. "5A10" for Type 1 diabetes) and the fully qualified name. This can enrich our IR (e.g., tagging a condition mention with `meta.icd11_code`). (For **adapter integration**, secure storage of the OAuth credentials and token refresh logic will be needed, although for open use the token can be long-lived.)

(The above APIs have a global/U.S. focus and rich structured data. All are free and open-access, with no paywalled content.) Each of these will be incorporated via a dedicated adapter or integrated service, as described next.

Section B: Python Libraries for Open Research Article Discovery and Ingestion

To ingest open-access scientific literature, we leverage several libraries and APIs that simplify finding and retrieving papers. Below we highlight tools centered around **OpenAlex (pyalex)** and related services, describing how to use them to locate open-access articles, get metadata/full-text, and integrate into our ingestion pipeline:

OpenAlex via PyAlex – Open Research Metadata

- **About pyalex (OpenAlex):** *OpenAlex* is a comprehensive index of scholarly works, authors, journals, and more, with open metadata. **PyAlex** is the official Python library for OpenAlex, which wraps the OpenAlex REST API. It allows searching for articles by keywords, filtering by fields (e.g., year, open access status), and retrieving rich metadata. OpenAlex covers >200 million works, including crossref, bibliographic info, and Open Access status.
- **Finding Open-Access Articles:** PyAlex supports filtering works by `is_oa` flag and `oa_status`. For example, one can query all works in 2020 that are open access:

```
from pyalex import Works
results = Works().filter(publication_year=2020, is_oa=True).get()
```

  This uses the OpenAlex API to return works from 2020 where an OA full-text is available [36]. PyAlex handles pagination transparently, so `results` will iterate through the records (25 per page by default). We can also add keyword search: `Works().search("machine learning cancer").filter(is_oa=True, publication_year=2023).get()`. The OpenAlex API also allows complex boolean filters (AND/OR on fields) – pyalex provides chainable `.filter()` calls for that [36].
- **Retrieving Metadata & Full-Text Links:** Each Work returned by pyalex is a dictionary-like object containing metadata such as title, authors, abstract, venue, DOI, and Open Access info. For example, `work["doi"]` gives the DOI, and `work["open_access"]` provides OA details [37]. Specifically, `open_access` includes `is_oa` (bool), `oa_status` ("gold", "green", etc.), and an `oa_url` if available [38]. PyAlex can even fetch the abstract text easily (`work["abstract"]` property formats the abstract from the inverted index) [39]. However, OpenAlex **does not host full-text PDFs** – instead it links to them. Typically, for an OA work, `best_oa_location` (available in the raw API data) gives a URL to a PDF or HTML version [40] [41]. To get full-text, we use these URLs (often via Unpaywall as discussed below).
- **Integration into Adapter SDK:** We will create an **OpenAlex Adapter** using pyalex. In an ingestion job, this adapter can take a query (topic or DOI list) and use pyalex to fetch relevant works. For each returned work, we normalize the metadata into our IR schema: e.g., create a `Document` with fields like `title`, `authors`, `venue` (journal/conference), `year`, and identifiers (`doi`, OpenAlex ID). We also capture OpenAlex's IDs for authors and journals, which can later link to knowledge graph

entities (authors, institutions). Crucially, the adapter will record the `is_oa` flag and any `oa_url`. If an OA URL for the PDF is present, we pass it along for full-text retrieval (either automatically or as a subsequent ingestion step). The adapter can also use the DOI to cross-link with other services (Crossref for additional metadata or direct Unpaywall lookup).

- **Rate Limits & Notes:** OpenAlex API allows 100k requests/day and 10 requests/second [42]. PyAlex internally respects these limits and we set our contact email in `pyalex.config.email` for polite usage [43]. Known limitations: OpenAlex's snapshot is updated monthly (so newest papers appear with slight delay), and very large result sets require paging through (pyalex handles this but we must be mindful of performance). We'll implement caching for repeated queries (e.g., storing results of a known DOI to avoid re-fetch).
- **Example Python Usage:**

```
from pyalex import Works
Works().search("COVID-19 AND long-term effects")\
      .filter(is_oa=True, publication_year=2021)\
      .select(["id", "doi", "title", "open_access"])\
      .get()
```

This returns metadata for OA works in 2021 about COVID-19 long-term effects, including each work's OpenAlex ID, DOI, title, and OA status. The adapter will convert each such result into our normalized IR (and then trigger full-text download if `open_access.oa_url` is provided).

Unpaywall API/Unpywall – Open-Access PDF Links

- **About Unpaywall:** Unpaywall is a database of open-access status for scholarly articles, keyed by DOI. It finds legal OA full-text (in repositories or publisher sites) for articles. Unpaywall has an easy REST API (no auth key required, just an email parameter) [16]. There's also a Python client (`unpywall`) for convenience [44].
- **Finding OA Full-Text:** Given a DOI, the Unpaywall API returns a JSON with metadata including whether the article is OA, the license, and up to several locations where full-text is available. Critically, it provides direct links to PDFs or HTML. For example, querying DOI *10.1038/nrn3241*:

```
curl "https://api.unpaywall.org/v2/10.1038/nrn3241?email=YOUR_EMAIL"
```

returns JSON indicating if the article is free, and fields like `best_oa_location` (with `url_for_pdf` and `license`). A typical response has `"is_oa": true` and e.g. `"best_oa_location": {"host_type":"publisher","url":"https://.../nrn3241.pdf","url_for_pdf":"https://.../nrn3241.pdf","license":"cc-by"}`. The adapter will extract the `url_for_pdf`.

- **Retrieving Metadata/Full-text:** Unpaywall's data overlaps with OpenAlex (since OpenAlex incorporates Unpaywall data for `oa_status` and `oa_url`). However, Unpaywall can sometimes provide additional locations or updated info. We will use Unpaywall primarily to **get a reliable PDF URL** for a given DOI. The metadata it returns (like journal name, year, etc.) is similar to Crossref; we'll primarily rely on OpenAlex/Crossref for metadata and use Unpaywall for access links and license info. Unpywall (the Python library) can bulk query DOIs and get a pandas DataFrame of results, which might be useful for batch operations.

- **Integration into Ingestion Flow:** The **Unpaywall integration** will be used in conjunction with other sources: e.g., after the OpenAlex adapter yields a set of DOIs that need full text, we call Unpaywall for each DOI to get the PDF link. This can be done via a small helper (we may not need a full standalone "adapter" with API endpoint, since Unpaywall is an enrichment step). For efficiency, we might integrate Unpaywall calls inside the OpenAlex adapter: for each work that is `is_oa`, verify or fetch `url_for_pdf`. Alternatively, we could queue a separate **Unpaywall adapter** job that takes a DOI list and returns a list of PDF URLs and licenses. In either case, the orchestration will combine the metadata (from OpenAlex) with the access URL (from Unpaywall). The license info (e.g. CC-BY) from Unpaywall will be stored in our IR (e.g., in `Document.meta.license`).
- **Rate Limits & Notes:** Unpaywall requests require an email parameter and should be limited to ~100,000 calls per day [16]. This is typically plenty; if we need more, Unpaywall offers dataset dumps. We will include our contact email in queries. The Unpaywall API is very fast for single requests; for bulk, we must throttle (the Python client may handle some batching). One limitation: Unpaywall only covers articles with DOIs; if a work lacks a DOI, it won't have Unpaywall data (though OpenAlex usually assigns DOIs to most works). Also, occasionally `url_for_pdf` may be null if only an HTML version is available – our pipeline will handle both PDF and HTML ingestion (perhaps using our PDF parsing if PDF link exists, or HTML ingestion pipeline if not).
- **Example Usage:** Using the Python `requests` library directly:

```
import requests
doi = "10.7717/peerj.4375"
res = requests.get(f"https://api.unpaywall.org/v2/{doi}?
email=me@example.com").json()
if res.get("is_oa"):
    pdf_url = res.get("best_oa_location", {}).get("url_for_pdf")
    print(f"PDF URL: {pdf_url}, License: {res.get('license')}")
```

This would output, for example, a PDF link and license for a PeerJ article (which might be CC-BY [38]). In our adapter, we'd use this `pdf_url` to download the paper and then feed it into the parsing pipeline (MinerU PDF processing or direct text extraction if HTML).

Crossref API (REST) – Scholarly Metadata

- **About Crossref:** Crossref's REST API provides bibliographic metadata for publications indexed by Crossref (primarily journal articles, conference proceedings, etc. with DOIs). It's a key source for article titles, authors, affiliations, references, and DOI resolution. Crossref metadata often complements OpenAlex. There are Python libraries like `habanero` (in R) and `crossrefapi` in Python, but one can easily use `requests` to call Crossref.
- **Finding Articles (Open Access Filter):** Crossref API allows searching by title, author, etc. Example:

```
curl "https://api.crossref.org/v1/works?
rows=5&query.title=cancer+immunotherapy"
```

returns metadata for works with "cancer immunotherapy" in the title. Crossref itself doesn't label open-access status explicitly, but it does provide license URLs if the publisher has deposited them. We can filter by existence of a license: e.g., `filter=has-full-`

`text:true,license.url:*creativecommons*` might find OA articles (if they have a Creative Commons license on record). However, this filter can be hit-or-miss. In practice, we rely on Unpaywall or the presence of a Crossref license URL to identify OA.

- **Retrieving Metadata and Full text:** Crossref returns JSON with fields like `title`, `author` list (with names and ORCIDs if available), `published-print` date, `container-title` (journal name), `reference` list (if the publisher provided references), and sometimes a `link` array with URLs. The `link` array can include full-text links; specifically, Crossref's metadata may have `URL` for the DOI landing page and license information. If a work is open, a `license` field with a URL and `content-version` may be present. We will parse these fields to enrich metadata: e.g., populate `Document.meta.journal`, `meta.authors` (with author names/ORCIDs), `meta.doi`. If references are provided, we may store them as linked Document references in our graph (though for v1 we might skip full reference ingestion). For full text retrieval, Crossref is generally not used (it doesn't serve PDFs), but it can direct us to publisher sites or tell us if an article has a CC license.
- **Integration & Adapter Flow:** We plan a **Crossref Adapter** mainly for two cases: (1) to retrieve high-quality metadata for a known DOI (e.g., when ingesting by DOI, Crossref gives us authoritative titles/authors which we can cross-verify with OpenAlex), and (2) to find references or citations for an article (Crossref's `/works/{DOI}/transform` or the `reference` list). The adapter will likely use the Crossref `/works/{doi}` endpoint to get a JSON for that DOI. The returned data will be merged into our Document's meta. If OpenAlex was already used, Crossref might be supplementary (OpenAlex itself sources from Crossref). We'll also ensure to include a `mailto` or User-Agent email to be in the polite pool [43], as Crossref allows 50 requests/sec per IP for identified ("polite") clients [45].
- **Rate Limits & Notes:** Crossref's public API is free but subject to rate limiting (50 req/sec/IP in polite pool) [45]. For our use (which is likely under that), this is sufficient. We should include `mailto` in headers as they request, to avoid being blocked. One limitation: not all Crossref records have complete metadata (e.g., some may lack references or ORCIDs depending on publisher deposition). Also, Crossref doesn't store abstracts (whereas OpenAlex sometimes has an abstract or via other sources). So we may still need other services for abstracts or full text.
- **Example Usage:** Using the `crossrefapi` Python library (if installed) or direct requests. For example, with the `requests` library:

```
import requests
doi = "10.1101/2020.04.01.20050502"
res = requests.get(f"https://api.crossref.org/v1/works/{doi}",
                   headers={"User-Agent":"MedKG/1.0
(mailto:me@example.com)"}).json()
title = res['message']['title'][0]
print(title, " - License:", res['message'].get('license', [{}])
[0].get('URL'))
```

This might print the title of a medRxiv preprint and its license URL (if any). In our system, we would use the Crossref data to ensure the title/author metadata is accurate and possibly detect if the license indicates open access (e.g., license URL contains creativecommons). The adapter then proceeds to download the PDF via Unpaywall or directly if a link is provided.

CORE (core.ac.uk) API – Open Access Repository Aggregator

- **About CORE:** CORE aggregates research papers from thousands of institutional repositories and open-access journals, providing both metadata and often full-text PDFs. The CORE API (v3) allows searching papers and fetching full-text content when available. This is a valuable source for open-access full-text (especially Green OA copies).
- **Finding Open-Access Articles:** By definition CORE's corpus is composed of open-access content. We can query by keywords, authors, etc. For example, CORE API allows queries like `title:"gene therapy" AND year:2022`. The API requires an API key (free registration) [46] and uses a JSON query syntax. We might use CORE to find OA versions of articles that are otherwise behind paywalls (CORE often has repository PDFs).
- **Retrieving Metadata/Full text:** CORE's response for an article includes metadata (title, authors, DOI if known, repository info) and crucially often a **downloadUrl** for the PDF of the article. It also provides a unique CORE ID for each work. We can use the `/articles/get/{CORE_ID}/download` endpoint to fetch the PDF directly (provided in their docs). The metadata includes fields like `description` (abstract), `fullText` (possibly the text if we request it), and `topics`. If our system needs the actual full text and Unpaywall didn't yield a PDF (or we want an alternate source), CORE can be a backup: e.g., if Unpaywall says an article is Green OA in a repository, CORE likely has that repository's copy. We will integrate CORE for **full-text ingestion** by using the `downloadUrl`.
- **Integration Notes:** We'll add a **CORE Adapter** that can take a query or DOI. In practice, we might primarily use CORE by DOI: CORE provides a "search by DOI" endpoint which returns the CORE record for that DOI (if available). Our adapter can do: if Unpaywall finds an OA location in a repository, use CORE to fetch the PDF from that repository via CORE's unified API (saves us writing custom scrapers for each repository). The adapter will use the API key and respect rate limits (CORE v3 suggests ~5 requests per 10 seconds for single-item requests [47]). We will need to queue these calls since full-text downloads are heavier. Once retrieved, the PDF goes through our PDF parsing pipeline. If the API also gives text, we could use it directly, but usually PDF is safer to ensure we capture figures/tables via MinerU.
- **Rate Limits & Policies:** CORE's free tier allows **up to 5 requests per 10 seconds** (i.e., 0.5 rps) for single-item calls [47]. It also supports batch requests (1 batch/10 seconds). This is a bit slow; if we need to ingest many papers, we may batch or request a higher quota. CORE encourages contacting them for higher limits if needed. Also, CORE's data is only as comprehensive as the repositories they harvest – some very new papers might not be in CORE yet.
- **Example Usage:** Using Python (pseudocode):

```
import requests
headers = {"Authorization": "Bearer <CORE_API_KEY>"}
doi = "10.1186/s13059-021-02451-0"
search_url = f"https://api.core.ac.uk/v3/search/works?query=doi:{doi}"
meta = requests.get(search_url, headers=headers).json()
core_id = meta['results'][0]['id']  # CORE work ID
pdf_url = meta['results'][0]['fullTextUrl'] or meta['results'][0]
['downloadUrl']
pdf = requests.get(pdf_url, headers=headers).content
```

Here we searched CORE for a specific DOI and retrieved the PDF content. Our CORE adapter would perform these steps internally. After getting the PDF, it would hand it off to the ingestion pipeline

(storing it and triggering MinerU for parsing). Metadata from CORE (if richer than Crossref/OpenAlex for that item) can also be used – e.g., some repository records include an abstract or additional subjects that we can map to our `Document.meta.keywords` or similar.

Semantic Scholar API – Academic Graph & Full Text (limited)

- **About Semantic Scholar:** Semantic Scholar (by AI2) offers an API to its academic graph, including paper metadata, citations, and (for some papers) full text snippets. The API has a free tier requiring an API key (intro rate ~100 requests/5min, higher for partners) [48]. It can be useful for reference mining and for cases where we want to gather citation networks or when other sources fail to get data.
- **Using it for Open Access discovery:** Semantic Scholar's data includes an `isOpenAccess` flag and often an `openAccessPdf` field (with a PDF URL and source) for papers in their corpus. So we can query by paper ID or DOI and check if `isOpenAccess=True`. For example, using their Graph API:

```
curl -H "x-api-key: <KEY>" "https://api.semanticscholar.org/graph/v1/paper/DOI:10.1038/nrn3241?fields=isOpenAccess,openAccessPdf"
```

  would return whether that DOI is open and a link if available. This overlaps with Unpaywall's info, but could serve as a double-check or alternative route (e.g., Semantic Scholar might have PDFs for some sources not covered elsewhere). Also, one can search by keyword via their API (`/paper/search?query=...`) though the free tier limits make large searches tricky (100/5min) [49].
- **Retrieving metadata/full text:** The Semantic Scholar API returns comprehensive metadata: title, authors (with IDs), abstract (sometimes), venue, and citation counts. It can also return the paper's reference list and citations (useful for graph building). However, full text is not directly given except for some PDF links. They have a experimental S2 Data where one can get structured text for some open papers, but that's not broadly available via API without special access. We will primarily use Semantic Scholar to **augment metadata** and possibly to gather references or citation counts for context. For example, if we ingest a paper's metadata from OpenAlex/Crossref, we could call S2 to get its citation count or influential citations for knowledge graph nodes.
- **Integration Notes:** A **Semantic Scholar Adapter** will be included for targeted enrichment. For instance, after ingesting a batch of papers, we could use their DOIs to fetch extra info from S2: such as the list of references (which we can then attempt to ingest if needed), or to identify highly cited papers we might have missed. In the ingestion job flow, this might be an optional step (since OpenAlex already covers a lot of this data). But Semantic Scholar's strength is in network analysis – we might use it to pull **citations graph** around a topic (e.g., find other papers that cite a key paper, which could hint at relevant literature). Rate limiting will require us to throttle (100 requests/5 min = ~20/min without special approval) [50]. We might batch DOIs to reduce calls (though S2's API mostly is one-paper-at-a-time for detailed info). If needed, we can apply for higher rates by describing our project (they allow 100 rps for partners in some cases [50]).
- **Limitations:** The free tier is fairly restrictive in throughput. Also, the coverage of full text links might not exceed Unpaywall's. Another limitation is that S2 may not have every new paper (there can be delays, though it's quite comprehensive). We will use it sparingly for data that adds value beyond other sources (like citation contexts or author influence metrics).
- **Example Usage:** Using their official `semanticscholar` Python package:

```
from semanticscholar import SemanticScholar
sch = SemanticScholar(api_key="YOUR_KEY")
paper = sch.get_paper("DOI:10.7717/peerj.4375",

fields=["title","isOpenAccess","openAccessPdf","citationCount","references.title"])
print(paper.title, paper.citationCount, paper.isOpenAccess,
paper.openAccessPdf)
```

This would print the title, number of citations, OA flag, and possibly a PDF link for that paper. The `references` field fetched shows titles of references, which we could decide to fetch as well. In our pipeline, such data could populate graph edges (Paper A cites Paper B). We will likely defer full citation graph integration to a later phase, but keep the hooks ready (i.e., store reference DOIs in our IR for possible later ingestion).

Europe PMC API – Literature and Preprints (Open Access Fulltext)

- **About Europe PMC:** Europe PMC is a repository integrating PubMed records, PubMed Central open-access full texts, plus additional European research funders' content and bioRxiv/medRxiv preprints. The API provides both metadata and access to open full texts (where available in PMC). It's a prime source for biomedical full-text articles (especially those in the PMC OA subset).
- **Finding Open Access Articles:** Europe PMC allows filtering search queries to only return free full text. For example: using their RESTful search:

```
query: "cancer immunotherapy",
filter: OPEN_ACCESS:Y
```

will return results that have free full text [51] [52]. One can also specify `AND SRC:PMC` to get only those in PubMed Central. The API's search syntax is powerful (supports boolean queries, MeSH tags, etc.). We can also query by DOI or PMCID directly.

- **Retrieving Metadata and Full-text:** The Europe PMC `search` endpoint returns JSON with metadata fields like title, journal, author, abstract, and identifiers (PMID, PMCID, DOI). If an article has full text in Europe PMC, we can get it via the `retrieve` endpoint in formats like XML or PDF. For example, if a result has `pmcid = PMC123456`, we can fetch `https://www.ebi.ac.uk/europepmc/webservices/rest/PMC123456/fullTextXML` to get the JATS XML of the article. This is incredibly useful because it gives structured full text (which our ingestion can parse without needing OCR or PDF processing). Our system already handles JATS for PubMed Central (as seen in the blueprint for PMC OA XML ingestion) [53]. Europe PMC also provides a bulk download of open access subset, but for dynamic queries the API is fine.
- **Integration Notes:** We will extend the existing **PMC Adapter** (or create a EuropePMC adapter) to utilize this. The blueprint's current PMC ingestion expects a known PMCID to fetch JATS [54]. We can enhance it to: given a DOI or search query, use Europe PMC API to find if the article is in the open-access subset. If yes, retrieve the full text JATS. Concretely, in the adapter flow:
- Accept input DOI or article title.
- Call Europe PMC `search?query=DOI:{doi}`. If found and has `isOpenAccess:true`, get the PMCID.

- Call the fullText fetch to get XML. Parse XML to IR (the blueprint already defines how to parse JATS IMRaD structure to `Document` and `Block` elements with sections like Introduction, Methods, etc. [54]).
- If the article is not in PMC, fall back to other methods (PDF via Unpaywall).
  This multi-step logic will increase success in obtaining full text. Europe PMC also covers preprints: e.g., bioRxiv entries often have full text (as XML or PDF) in their system. We can similarly retrieve those.
- **Rate Limits & Usage:** The API is free and doesn't require a key for moderate use. They request not to exceed 30 requests/second. Our usage will be much lower. We should include an `email` or `user-agent` as well. The search results are paginated (default 25 per page, up to 100). For large result sets, they provide a cursor-based pagination. We might not need that except for broad topic searches. A known limitation: some articles might be in EuropePMC as abstracts only (if not open access). The adapter should check the `fullTextUrlList` or `hasPDF` flags in the result.
- **Example Usage:** Using their REST API via requests:

```
import requests
query = 'heart disease OPEN_ACCESS:Y'
url = f"https://www.ebi.ac.uk/europepmc/webservices/rest/search?
query={query}&format=json&resultType=core"
resp = requests.get(url).json()
for hit in resp['resultList']['result']:
    print(hit['title'], hit.get('pmcid'), hit.get('doi'),
hit.get('isOpenAccess'))
```

This prints titles of open access results and their PMCID/DOI. For any with a PMCID, we can then do:

```
pmcid = hit['pmcid']
xml = requests.get(f"https://www.ebi.ac.uk/europepmc/webservices/rest/
{pmcid}/fullTextXML").text
```

and then parse the XML. In our integration, this process will be encapsulated. The net effect: the ingestion pipeline can obtain **full text without even using a PDF** for many articles (structured XML yields higher quality parsing). This complements the PDF-based approach (and avoids calling MinerU when we have XML). The adapter will therefore route either to "Auto-pipeline" (for XML) or "Two-phase PDF" as needed, similar to how PMC vs PDF is handled in the blueprint [53].

(Additional tools like the NCBI E-utilities for PubMed could be mentioned, but since we focus on open access content, we rely on PMC and other OA sources instead of just abstracts from PubMed. The above libraries collectively enable discovery of OA literature and retrieval of content.)

Section C: Modifications to the Blueprint Architecture to Support These Sources

In light of the above integrations, we will update the engineering blueprint for the **multi-protocol API gateway and ingestion/orchestration system**. This involves adding new adapters, extending the

orchestration logic for multi-source enrichment, updating our data schema to capture publication metadata, and modifying the OpenAPI specification and directory structure accordingly.

C.1 New Adapters for Biomedical APIs and Libraries

We will introduce several new **adapters** (each encapsulating the logic to fetch and parse data from a source) in our system. Each adapter follows the standard interface (`fetch -> parse -> validate -> write`) as defined in the blueprint [55]. Key additions include:

- **ClinicalTrialsAdapter:** (If not already present) – fetches trial data from ClinicalTrials.gov API. Accepts either a specific NCT ID or a search query. *Implementation:* For individual mode, call `/studies/{NCT_ID}` endpoint to get JSON for that trial [56]. For search mode, call `/studies?query.xxx=...` with appropriate filters. Parse the JSON into our IR: one `Document` per trial, with trial metadata (title, status, phase, etc.) stored in `Document.meta`, and detailed fields (eligibility criteria, outcomes, etc.) as `Block` elements (or nested sections). This adapter is an **auto-pipeline** (JSON in, structured IR out) and will mark the `Document.source` as "ClinicalTrials.gov". It will be wired to the existing `/ingest/clinicaltrials` API path [57], replacing or augmenting the stub there. It will also handle pagination if a query returns many trials (looping until done or a configured max).

- **OpenFDAAdapter(s):** We will likely have separate adapters for **Drug Adverse Events** and **Drug Labels** (and possibly devices). For example, `OpenFDADrugEventAdapter` will accept a drug name or substance and call the `drug/event` endpoint, retrieve up to N reports (with pagination via `skip`). It will produce a `Document` for the query (or one per report, depending on design). We might treat the entire set of results as one IR Document containing multiple Blocks (each Block could represent a case report or a summary of reactions). Alternatively, each FAERS case could be its own Document. We need to consider downstream usage – likely we aggregate by drug. Similarly, `OpenFDALabelAdapter` will fetch the SPL JSON for a given SetID (DailyMed setid) or do a search by drug name. It will parse key sections from the label (Indications, Dosage, etc.) into our IR Blocks (with `meta.loinc_section` tags as per blueprint [58]). These adapters tie into the existing ingestion flows: e.g., the blueprint had DailyMed SPL and openFDA devices already listed [59]. We'll extend those: the DailyMed adapter remains for labels (but could use openFDA's label end-point as an alternative data source). And for devices, we already have GUDID/UDI adapter [60]; we may add openFDA device recall/event as well.

Adapter configuration: Each new adapter will be described in YAML (or code). For instance, we'll add YAML entries under an **adapters/** directory (if using config files) like:

```
adapters:
  clinicaltrials:
    type: rest
    entry_point: med.adapters.clinicaltrials:ClinicalTrialsAdapter
    params: {...}
  openfda_drug_event:
    type: rest
    entry_point: med.adapters.openfda:DrugEventAdapter
```

```
    rate_limit: 240 per_minute
    params: {api_key_env: OPENFDA_API_KEY}
```

This indicates how the orchestration can invoke them. Each adapter class will handle constructing the URL and parsing. We include any needed API keys via config (e.g., OPENFDA_API_KEY from environment).

- **OpenAlexAdapter:** This adapter will integrate with the pyalex library. It supports two modes: (a) **Search mode** – given a query (topic keywords, filters like year or author), retrieve a list of relevant works; (b) **Lookup mode** – given a DOI or OpenAlex ID, retrieve that work's metadata. The adapter's `fetch()` will utilize pyalex (or direct API calls) to get results. Its `parse()` will transform each OpenAlex Work into our IR format: it will likely create one `Document` per work (article). Each Document's content initially might just be the abstract (as Blocks) and metadata (title, journal, etc.). It will also store references if provided (OpenAlex gives a referenced-by count and IDs of references, but not the full list of references – we might skip deep references for now). Crucially, the OpenAlex adapter will **trigger follow-up actions** for full-text: see Orchestration below. If `is_oa=true` and an `oa_url` is present for a work, the adapter can either directly fetch the PDF in `fetch()` (making it a multi-step adapter) or more cleanly, emit a partial IR (with metadata only) and signal the orchestrator to fetch the PDF via another adapter. We favor the latter (keeping adapters single-responsibility). So OpenAlexAdapter likely ends after writing metadata IR (documents with perhaps just abstracts), and then enqueues the DOI/URL for the next stage.

- **UnpaywallAdapter:** This could be a lightweight adapter that takes a DOI (or list of DOIs) and returns the best OA link. It might not expose a public API endpoint (since it's more of an internal helper), but we can implement it as a service in the adapter framework for consistency. For example, after OpenAlexAdapter completes, the orchestrator could call UnpaywallAdapter for each DOI that needs a PDF. UnpaywallAdapter's `fetch()` will call the API for the DOI and its `parse()` returns a small IR (or even just metadata update). We might not create a full Document for this; instead, we could update the existing Document's meta. A design choice: we can have the **ingestion job context** carry over the DOI and attach the PDF URL to the Document (since our ingestion pipeline currently is batch-oriented per adapter). Another approach is to incorporate Unpaywall logic inside OpenAlexAdapter's code (to reduce overhead), but that blurs separation. For clarity, we treat it as another adapter in code, but orchestrate it as part of a *multi-source workflow* (see next section).

- **CrossrefAdapter:** Similar to OpenAlexAdapter's lookup mode. Given a DOI, fetch Crossref metadata (JSON) and merge. This adapter could be invoked on-demand (for example, if OpenAlex is missing some info like reference lists or if we ingest something by PMID, we could use Crossref to get DOI). CrossrefAdapter will map fields to IR `Document.meta`: e.g., `meta.journal`, `meta.volume/issue`, `meta.page`, `meta.authors` (list of author names or even nested author objects if we choose to). Many of these fields were in the design implicitly, but we will explicitly add them to the schema (see Schema updates below). If references are included, we may attach them in an intermediate field (perhaps store as `meta.references = [DOI1, DOI2,…]` for potential later use). Initially, we might not use CrossrefAdapter for all entries since OpenAlex covers most metadata, but it's available for completeness or specific tasks (like referencing formatting or if we trust Crossref more for certain data).

- **COREAdapter:** Will handle full-text retrieval. It might accept either a search query (less likely, since we will use it mostly for known documents) or a DOI. In our flow, we'd use COREAdapter if Unpaywall gives a repository link but not a direct PDF – CORE often has the PDF. The adapter will use the CORE API (with our API key) to find the CORE ID by DOI, then get the download URL for PDF. We might integrate this with our PDF ingestion step – e.g., instead of directly downloading via Unpaywall link (which might be an HTML landing page), call CORE to get the PDF if available. The COREAdapter would output the raw PDF bytes (which then go to MinerU parsing). This suggests the COREAdapter might skip the normal IR writing (since IR is produced after parsing PDF). Instead, COREAdapter could be part of the **fetch stage of a PDF adapter**. Another design: treat CORE as part of the Unpaywall/Fulltext resolution process rather than a separate pipeline step. For now, we list it so we have the capability: e.g., a `GET /ingest/core` (internal) that given a DOI returns a PDF stream or text. In code, this can be a utility.

- **SemanticScholarAdapter:** This adapter will be used for enrichment tasks like adding citation data. It won't be part of the initial ingestion for every paper (to save time and API calls), but we can have it in our toolkit. For example, after ingesting a set of documents, we could run a job that uses SemanticScholarAdapter to fetch each document's citations count or references. The adapter takes a DOI or S2 Paper ID and calls the S2 API. It then could update the corresponding Document node in our graph or index with the new info. Since our IR is mostly focused on content extraction, citation metadata might be handled at the KG layer instead (the blueprint's later steps). We will therefore integrate Semantic Scholar in the **mapping/extraction phase**: possibly using the `extract/map → KG` step to add citation edges. Nonetheless, the adapter lives in the codebase in case we need to fetch any data via S2 (like for a particularly important paper's references).

- **EuropePMCAdapter:** We will enhance the existing PMC adapter to utilize Europe PMC's search. In practice, this might be inside the PMC adapter logic: e.g., currently `/ingest/pmc` expects a PMCID input [54]. We can extend the API to allow DOI or PMID input: if DOI given, the adapter will query Europe PMC for that DOI to find if a PMCID exists (and if open access). If found, proceed to fetch the full text XML. So, changes include: updating the `/ingest/pmc` endpoint to accept either a PMCID or DOI in the request schema (update OpenAPI accordingly). The adapter code will handle the conditional lookup. Similarly, we might allow a search query input for broader ingestion (though likely we'll use OpenAlex for search, and then get PMCID for those DOIs). The net is a more flexible **PMCLiteratureAdapter** that covers multiple input forms.

Adapter Directory Structure: In the project repository, we will add new modules under `med.adapters`. For example:

```
med/
 └ adapters/
     ├ clinicaltrials_adapter.py
     ├ openfda_adapter.py   (with subclasses for DrugEvent, DrugLabel, Device)
     ├ openalex_adapter.py
     ├ unpaywall_adapter.py
     ├ crossref_adapter.py
     ├ core_adapter.py
```

```
    ├ semantic_scholar_adapter.py
    └ pmc_adapter.py   (extended for EuropePMC)
```

Each file contains the adapter class implementing `fetch/parse`, etc. We will also update any **adapter registry** the system uses (could be a YAML or a Python dict mapping source names to classes). This ensures the ingestion orchestration can discover and invoke these new adapters by name. For instance, the config might map `"openalex"` -> `OpenAlexAdapter`, `"clinicaltrials"` -> `ClinicalTrialsAdapter`, etc.

The **OpenAPI specification** will be updated to add new ingestion endpoints for these sources, as described next.

C.2 Ingestion Orchestration Extensions for Multi-Source Enrichment

Our orchestration engine (which coordinates the ingestion pipeline, Kafka jobs, and adapter invocation) will be extended to handle **multi-step ingestion flows** where necessary. The blueprint already distinguishes between auto-pipeline vs two-phase (for PDFs). Now we introduce flows that involve *multiple adapters sequentially*.

pyalex + Unpaywall multi-source enrichment: The prime example is literature ingestion by topic. The process will work as follows:

1. **Initial Metadata Fetch:** The orchestrator receives a request (via API or scheduled job) to ingest literature for a topic or DOI list. For a topic query, it will first invoke **OpenAlexAdapter** (Section B) with the query parameters. This adapter fetches metadata for relevant works and writes preliminary IR entries (Documents with metadata, possibly abstracts). Each such Document will have a placeholder indicating that full text is pending (for example, a flag in `Document.meta` like `"full_text_status": "pending"`). The adapter also emits the DOI and any OA URL for each work, likely in the Document's metadata. If an OA URL (particularly a PDF link) is directly available and we trust it, we could enqueue a PDF download directly. Often, however, we'll have a DOI and maybe an OA link that needs resolution.

2. **Enrichment Step – Full Text Retrieval:** For each Document from step 1 that is open access, the orchestrator will trigger a **PDF ingestion sub-flow**. This can be done in two ways:

3. **Option A:** The OpenAlexAdapter itself could enqueue a PDF fetch job in the ledger. For example, after writing the Document, it could add a ledger record `{doc_key, media_type:'pdf', status:'pdf_pending', uri:<PDF URL>}`. Then a worker (or a subsequent call) uses an existing PDF ingestion mechanism to fetch and parse it (similar to how PDF two-phase is done) [61]. We might integrate this with the MinerU pipeline by reusing the `med ingest pdf` CLI approach for each URL.
4. **Option B:** The orchestrator explicitly calls another adapter like **FullTextAdapter**. We can implement a FullTextAdapter that, given a DOI (and known to be OA), coordinates Unpaywall->CORE->download. But since PDF download/parsing is already handled by the PDF pipeline, we may just need to drop the PDF URL into that pipeline.

The likely approach: **Use the ledger two-phase flow**. We modify the ingestion flow: when an OpenAlex Document is written with a PDF URL, we automatically treat that as a PDF to fetch. Specifically, we can extend the **auto_inflight/auto_done** logic: after the auto step (metadata) is done, the orchestrator transitions to PDF fetch for those docs. This could be done by marking those Documents and then invoking an internal method equivalent to `med ingest pdf --uri <url> --doc-key <DOC_KEY>` for each. That command uses the existing PDF adapter (which downloads and stops before parse, leaving a ledger entry `pdf_downloaded`) [61]. The MinerU service then picks it up to produce IR from PDF (ledger `pdf_ir_ready` to `postpdf-start`) [62].

We will update the **orchestration code** to support this chaining: e.g., after ingesting via OpenAlexAdapter, iterate over results; for each, if `oa_url` exists: - Call the PDF fetcher (this could be synchronous or produce a message onto the `ingest.requests` topic with type "pdf"). We might create a small function that wraps the PDF adapter call for a given URL and doc ID. - Ensure the PDF content is stored (object store) and an event is emitted (just like manual PDF ingestion would). - Then the existing MinerU pipeline continues as normal (since it watches the ledger for `pdf_downloaded`). This effectively merges the two-phase pipeline with our new multi-adapter scenario.

1. **Metadata Merge:** When the full-text IR from the PDF is ready (i.e., MinerU has output JSON with content blocks), we need to merge it with the initial metadata Document. The blueprint's two-phase design already expects that after MinerU, we run `postpdf-start` which goes from IR to chunking, etc. [62]. We will adjust `postpdf-start` logic to **enrich the IR with any metadata from the earlier step**. Perhaps the easiest method: the metadata Document and the content Document could be the same logical `doc_id`. If we pass the same `doc_key` to the PDF adapter, then when we parse the PDF, we can incorporate the existing metadata. Concretely, the PDF parse step could load the stored preliminary IR (or get it via reference) and attach the content blocks to it. Alternatively, we ensure the metadata (title, authors, etc.) is present in the MinerU output by providing it as input context (though MinerU mainly cares about raw PDF -> text). More practically, we can simply take the MinerU output (which likely has minimal metadata beyond maybe the title extracted) and overwrite its metadata section with the richer metadata from OpenAlex. We can implement this in the **IR normalization step**: e.g., a small function after parsing that says "if this PDF's DOI matches a record we ingested from OpenAlex, merge the fields". This might happen in the `validate()` or just before `write()`.

Another approach is to generate the Document in IR only after full text is available: i.e., don't write a partial IR on metadata fetch, but store metadata temporarily, get PDF, then write a complete IR. However, writing partial IR is useful for tracking and if some papers have no PDF (metadata-only), at least they're indexed. We prefer to have at least metadata in the system even if full text is unavailable, so partial IR is fine. We just need to avoid duplication. To manage this: - Use a consistent `doc_id` (the blueprint suggests something like `{source}:{id}#{version}:{hash12}` for docs [63]). For OpenAlex sourced ones, we could define the doc_id to include the DOI or OpenAlex ID. For the PDF content, similarly. If those match, our system might naturally attempt to write to the same doc. We must ensure idempotent merge rather than treating content as separate doc. Possibly, we treat the metadata Document as final, and the PDF parse as updating it. - This may require changes in the `write()` step of adapters to support update vs insert. Alternatively, perform merge at KG loading: since DOI is a unique key, when loading into graph or index, combine records with same DOI. This could be simpler: e.g., our search index update could upsert documents by DOI. The knowledge graph ingestion (Neo4j) can merge nodes by DOI. So even if metadata IR and fulltext IR are two

separate JSON files, the integration layer can unify them. For safety, though, we try to keep it one Document in IR.

We will extend the **validation** or **writing** stage to handle merging if needed (e.g., the validate step could identify duplicate doc_ids and combine them, or drop the partial one once full one is present). This ensures that by the time we chunk and index, we have one coherent Document per article.

1. **Other Multi-adapter flows:** Similarly, if we ingest a **clinical trial** that mentions drugs or conditions, we might trigger adapters for those – e.g., after ClinicalTrialsAdapter outputs a trial, we could call RxNormAdapter to get RxCUI for each intervention name, or call Mesh/ICD adapter for condition mapping. This is a form of enrichment that the blueprint's extraction/mapping phase could cover (e.g., using ontologies listed in scope [64]). We will incorporate that by adding mapping rules: for example, an extraction rule could pick up condition strings in the IR and call an **ICD-11 adapter** to fetch codes, then attach them as structured tags in the KG. The orchestration can be event-driven: after a trial Document is ingested (and indexed), a separate **mapping job** (maybe via `mapping.events.v1` Kafka topic) could trigger these lookups for normalization. This is consistent with the plan to align with ontologies like SNOMED, ICD-11, RxNorm [64]. So essentially, the new adapters (RxNorm, ICD) will be invoked not directly via public API, but via internal workflow when needed. E.g., a background worker listens for new trial documents and then uses RxNormAdapter to annotate drug names. The results are then updated in the graph (this could be done by writing back to Neo4j or updating the IR with new `Block` entries labeled with codes). Our blueprint will note this extension in the *extraction & mapping stage*.

2. **Scheduling & Pipeline Config:** We will add configurations to control these flows. For example, a flag like `enable_multi_source_enrichment: true` in config can gate whether the orchestrator automatically does the Unpaywall/PDF step (similar to how feature flags exist for neighbor merges [65]). This gives flexibility to run in pure metadata mode vs full content mode. We will default it to enabled for our use-case (since full text is desired).

In summary, the orchestration is extended to handle *dependent adapter calls* and *two-phase content assembly*. The **job queue** will carry not just simple one-adapter tasks, but compound tasks: - e.g., a job message might specify: source: "openalex", query: "X", and a pipeline: `[openalex -> pdf]`. The orchestrator would execute openalex, then for each result, spawn the PDF ingestion. We might formalize this by allowing a **chained operation** in one API call (for instance, a new endpoint `/ingest/literature` that internally calls multiple adapters). Alternatively, the client (or our script) could call /ingest/openalex then call /ingest/pdf for each result. But an internal automation is better for large volumes.

We will update documentation to reflect that **ingesting literature now automatically involves OpenAlex + Unpaywall/CORE** to get full content when available.

C.3 Schema Updates for Publication Metadata

To accommodate the richer metadata from these new sources, we will extend the IR schema (and downstream index/graph schemas) to include additional fields, particularly for publications. New or extended fields include:

- **Document.meta.source** (already implicitly present): we'll ensure values like `"OpenAlex"`, `"Crossref"`, `"PMC"`, etc., are set appropriately so we know which pipeline provided the data. This is useful for provenance and debug (the blueprint's design emphasized provenance/citations for every piece of info).

- **Document.meta.title, meta.authors, meta.journal (venue):** We explicitly add `title` (string) if not already, and an `authors` list (each author could be represented as sub-object with name, and optionally ORCID or OpenAlex author ID). For example:

```
"meta": {
    "title": "Example Article",
    "authors": [
       {"name": "Doe, John", "orcid": "0000-0001-2345-6789"},
       {"name": "Smith, Jane"}
    ],
    "journal": "Journal of Examples",
    "volume": "12", "issue": "3", "pages": "45-60",
    "year": 2021,
    "doi": "10.xxxx/abcdef",
    "pmcid": "PMC1234567",
    "license": "cc-by",
    "oa_status": "gold"
}
```

  Many of these fields were not explicitly in the initial blueprint IR (which was more focused on content blocks). We add them to capture bibliographic info. The adapters will populate these from OpenAlex/Crossref/Unpaywall as available. The `doi` field is key for deduping and linking. `pmcid` or `pmid` can also be stored if known (e.g., if OpenAlex lists a PMID or if we got data via PMC). `license` and `oa_status` come from Unpaywall/OpenAlex to indicate openness. We also might include `concepts` or `keywords` if OpenAlex provides subject classifications – OpenAlex gives Fields of Study and concepts with scores; we can put top concepts in `meta.topics` (array of concept names or IDs). This could aid the knowledge graph (aligning with ontologies).

- **IR for Trials and others:** For ClinicalTrials, we similarly ensure the schema can hold structured trial info. Likely `Document.meta` for a trial document will have fields like `nct_id`, `status`, `phase`, `enrollment`, etc. (some of these were in the table of blueprint under "Identifiers extracted" [66]). We will formalize that. For example:

```
"meta": {
  "nct_id": "NCT04267848",
  "status": "Recruiting",
  "phase": "Phase 3",
  "study_type": "Interventional",
  "condition": ["Melanoma"],
  "interventions": ["Drug: Pembrolizumab"],
  "sponsor": "National Cancer Institute",
  "start_date": "2020-01-01",
  "completion_date": "2023-06-01"
}
```

and so on. These meta fields allow easier querying in the KG or search index (e.g., find all trials in Phase 3 for Melanoma, etc.). The blueprint already considered many of these, but we ensure consistency (maybe adopting the same field names as the ClinicalTrials.gov schema).

- **New Intermediate Reference (IR) fields:** If we decide to capture reference lists from Crossref or Semantic Scholar, we might add a field in Document, say `Document.references` which could list DOIs or other identifiers of cited works. For now, we may not use it heavily, but structuring it now allows later graph construction of citation networks. Similarly, for each Document (paper) we could have `citations_count`. These might be stored in meta as well. E.g., `"meta": { "references": ["10.1001/jama...","10.1136/bmj..."], "citation_count": 27 }`. The knowledge graph ingestion can then create edges for these references.

- **Adapter Output Normalization:** Each adapter's parse will yield JSON that fits this schema. We may create a **schema mapping** layer for consistency. For instance, OpenAlex gives authors as a list of objects with name and id; Crossref gives authors similarly. We unify those into the `authors` structure above. The blueprint's validation step (using Pydantic or JSON schema [67]) will be updated to include these new fields and enforce types. We will extend `config.schema.json` (if using one) accordingly. For example, add required field `meta.doi` for literature Documents, with pattern to validate DOI format, etc., and optional fields like `meta.license` must be one of known values or a URL.

- **Information Resource (IR) Types:** The blueprint's IR types included `Document`, `Block`, `Table` [68]. We maintain those. A journal article will be one `Document` with many `Block`s (each block could be a paragraph or a section chunk depending on chunking). For PMC XML, the parser likely creates `Block`s per section or paragraph already (with section titles like Introduction, Methods etc. possibly stored in block metadata). For PDF via MinerU, we get blocks of text and tables. We ensure that regardless of source, the text content ends up in Blocks for embedding & indexing. The meta fields we described sit at the Document level.

- **Graph Schema:** In the Neo4j graph, we may create new node labels like `Publication` (with properties DOI, title, year, etc.) and connect them to other entities (drugs, trials, concepts). The blueprint planned a coherent knowledge graph of all sources [69]. Now, with publications integrated, we will have edges such as `TRIAL -[:HAS_PUBLICATION]-> PUBLICATION` (for trial references),

`PUBLICATION -[:CITES]-> PUBLICATION` (for references, if we ingest them), `DRUG -[:MENTIONED_IN]-> PUBLICATION`, etc. We'll update the graph mapping rules accordingly (this might be handled in the extract/map phase by scanning IR text for mentions, or by explicit links like trial to publication from ClinicalTrials data's referencesModule [70]). Also, author information could form `Author` nodes linked to Publication, but that might be excessive for now; we might skip explicit author nodes in v1, keeping authors as strings.

C.4 OpenAPI Spec and API Gateway Updates

We will expose new API endpoints (or enhance existing ones) in our API Gateway for triggering these ingestions. In the **OpenAPI specification**, under the `/ingest` paths, we add:

- `POST /ingest/openalex` – Ingest open-access literature via OpenAlex (and downstream enrichment).
- **Request schema:** We define an `IngestOpenAlexRequest` containing either a search query with optional filters (fields like `query_string`, `filter_year`, etc.) or a list of identifiers (DOIs or OpenAlex IDs). For example:

  ```
  { "query": "lung cancer immunotherapy", "year": 2021, "limit": 50 }
  ```

  or

  ```
  { "doi_list": ["10.1000/xyz123", "10.1000/xyz456"] }
  ```

  We also allow a flag like `"full_text": true` to indicate whether to attempt full-text retrieval (default true).

- **Operation:** The backend will initiate the process described: call OpenAlexAdapter, then fetch PDFs, then index. This likely returns a 200 when the request is accepted and processed asynchronously (we might immediately ingest or put it in the Kafka queue). The response could be an `IngestBatchResponse` with status per DOI (like the blueprint's 207 multi-status response for batch [71]). For each item, status might be "Ingested" or "Queued". This is similar to how `/ingest/clinicaltrials` returns 207 if some IDs fail. We'll reuse that pattern.

- `POST /ingest/doi` (optional shorthand) – We might add a convenience endpoint to ingest a single DOI. This would essentially combine crossref/openalex + pdf. It could accept a JSON like `{ "doi": "10.xxxx/abcd", "fetch_full_text": true }`. Internally, it might call the same pipeline as /ingest/openalex (just for one identifier). The advantage is for a user who has a specific paper DOI to ingest. This endpoint could utilize CrossrefAdapter (to get metadata) if we choose, or OpenAlex (either is fine since by DOI both can retrieve; OpenAlex also gives the OA info). Possibly we use Crossref here to avoid unnecessary complexity, since Crossref's turnaround is quick for a single DOI and doesn't require parsing a large search result. Crossref would give basic metadata, then we'd call Unpaywall for PDF. We will document this in the OpenAPI if we include it.

- `POST /ingest/clinicaltrials` – Already in spec [57]. We will keep it but ensure it uses our ClinicalTrialsAdapter logic. The request schema was likely `IngestClinicalTrialsRequest` with an array of NCT IDs. We could extend it to accept a search query as well (e.g., `condition:` and `location:` filters). If so, we update the schema to allow a `query` field. When a query is provided, the service will fetch all matching trials (with some cap or pagination). If multiple NCT IDs are provided, it ingests those specific ones (as before).

- `POST /ingest/dailymed` – Already present for SPL by setid [72]. No change, except our OpenFDA Label adapter might be an alternate path to get the same data. We won't change the interface here; the user still provides a setid or NDC. Under the hood we might fetch from openFDA or from the DailyMed FTP as before. This can be abstracted in the adapter config (try one source, fallback to another).

- `POST /ingest/pmc` – Already present for PMCID [54]. We extend the request schema (`IngestPMCRequest`) to optionally allow DOI. We'll document that either `pmcid` or `doi` is required. If DOI given, the service will perform the EuropePMC lookup as described. From the API consumer perspective, they just hit this endpoint with a DOI and get the article ingested if open access. The responses remain similar (`IngestResponse` with doc id or error if not found/not OA).

- We may add `POST /ingest/mesh` **or** `/ingest/ontology` for concept normalization tasks (if we plan to let users trigger ontology mapping separately). But this might not be needed as an external API; it's more internal. Possibly an admin endpoint to refresh mappings or ingest an ontology (like ICD-11 codes). Given the scope, likely not needed now.

- No changes to `/chunk`, `/embed`, etc., except that now those processes will handle more Documents (from new sources) and possibly larger text (full articles). The blueprint's retrieval pipeline can remain the same (embedding, indexing) – it's source-agnostic. We might ensure that extremely large documents (like full-text articles) are chunked properly (which is already handled by `chunk` step, possibly using a max tokens or section-wise strategy).

- **Security & Auth:** All new endpoints will follow the same auth model – e.g., requiring an `ingest:write` scope token for protected use (as shown in spec for others [73]). The CORS and rate limiting on the gateway might be configured to allow these without impacting others.

OpenAPI components (schemas) for the new requests and responses will be added. E.g.:

```
components:
  schemas:
    IngestOpenAlexRequest:
      type: object
      properties:
        query: { type: string, description: "Search query for works
(optional)" }
        doi_list: { type: array, items: {type: string}, description: "List of
DOIs to ingest" }
        year: { type: integer, description: "Filter by publication year" }
```

```
        full_text: { type: boolean, default: true }
      oneOf:
        - required: [query]
        - required: [doi_list]
  # ... possibly IngestDOIRequest (similar to above simplified for one DOI)
```

Responses could reuse `IngestResponse` (for single) and `IngestBatchResponse` (for multiple, listing each DOI's status). If any item fails (e.g., DOI not found or no OA available), we return it with error info, similar to how partial success for ClinicalTrials was envisioned [71].

Adapter Directory and API Gateway: We will ensure the API gateway (if using a pattern like APIRouter or blueprint registration) knows about these new endpoints and maps them to the adapter invocations. The server might have a controller function that, for example, reads the OpenAlex request, pushes a message to the ingest queue (with payload specifying adapter = openalex, parameters = query or DOIs), and immediately returns an acknowledgment. Alternatively, it may perform it synchronously if quick. Given potentially many results, asynchronous (via Kafka ingest.requests topic) is better. The blueprint's orchestration mentions a Kafka topic `ingest.requests.v1` and `ingest.results.v1` for asynchronous processing [74]. We will use the same mechanism for the new adapters: publish tasks to `ingest.requests.v1` with appropriate payload (including a job ID, source, etc.), and the worker will process and then emit to `ingest.results.v1`. The API call could poll or just return a job ID and later one can GET result status. However, since existing endpoints seemed to return 200/207 directly, maybe a simpler approach is used (maybe they processed synchronously up to writing IR, which for one trial or one paper might be fine). For batch queries (like a topic that yields 100 papers), synchronous might be too slow for an HTTP call. Possibly we'll adopt an async model there. But for now, we can say it returns once ingestion is initiated, with a list of documents queued or ingested.

C.5 Adapter/Service Integration in the Architecture Diagram

In the system architecture, the new adapters will fit into the **Ingestion Service** component. The blueprint likely had a diagram or description dividing adapters by type. We are effectively adding: - *REST API Adapters* for new external APIs (ClinicalTrials, OpenFDA, OpenAlex, etc.). - *Library-based Adapters* (pyalex, unpywall – though those use REST under the hood too).
These all plug into the existing ingestion microservice. They utilize the HTTP client (with retry, etc.) as described in section 1.9 of blueprint [63]. We'll ensure to reuse the common HTTP logic (with backoff, timeouts as specified) for these external calls. For instance, openFDA and OpenAlex should both use the centralized HTTP client with standard timeouts, and adhere to our logging/tracing. Each adapter will produce logs and metrics (we'll add labels for source-specific metrics; e.g., number of papers ingested, time taken, etc., possibly exposed to Prometheus as counters per adapter). This ties into the observability section of the blueprint (Grafana dashboards can be extended to monitor these new sources – e.g., a panel for "OpenAlex ingestion count").

One more **infrastructure update**: we should store the large volumes of literature content. The object store (S3/MinIO) will now hold not just trial JSON and PDFs from guidelines, but also potentially hundreds of PDFs of papers. We confirm retention settings (365 days as per blueprint [75]) apply to all objects. No need to change, but we note increased volume – might need more storage allocation. The search index (OpenSearch) will also grow with article content – ensure indexing performance (we might consider using the "SPLADE index" for text as mentioned [76], which should handle it). We may consider splitting indices by

document type (e.g., one index for trials, one for publications) for query efficiency, but initially a unified index is okay since the retrieval pipeline can filter by source if needed.

Finally, **directory structure** adjustments: not only adapters, but we might create subdirectories for data source configuration. For instance, a directory `conf/` or `adapters/` with YAML for each. If using Singer or Camel-like frameworks was considered, we now essentially implemented connectors for each source. We'll maintain them in our codebase as above. The blueprint's mention of "universal connectors" (maybe in context/background) suggests this approach.

C.6 Example Workflow after Modifications

To illustrate how everything comes together, consider a user (Mercor analyst) wants all information on *"SGLT2 inhibitors in heart failure"*: - They call our API `/ingest/openalex` with query `"SGLT2 heart failure"`. The system (ingestion service) receives it and creates a job. The OpenAlexAdapter runs, retrieving, say, 20 relevant papers (some journal articles, some conference papers). It writes 20 Documents (with metadata + abstracts) to the IR store, and for each Document that is open access, it also fetches the PDF (through Unpaywall link or EuropePMC if available). Those PDFs get parsed by MinerU (GPU service), resulting in full text Blocks which update the Documents. The Documents then go through chunking, embedding, indexing (the pipeline automatically continues because for non-PDF, auto-pipeline did those steps; for PDFs, after MinerU we run `postpdf-start` to do chunk→embed→index) [62]. Now these articles' content is in the search index and their structured metadata in the KG. - Meanwhile, the user also calls `/ingest/clinicaltrials` with a search for condition=heart failure and intervention class=SGLT2. The ClinicalTrialsAdapter finds e.g. 5 trials, writes them as Documents with structured fields. These go through chunk→embed→index as well (trial descriptions chunked into criteria, outcomes, etc.). - Our mapping processes link things: the drug names (e.g., "dapagliflozin") in trials are normalized via RxNormAdapter to RxCUI, and we create a `Drug` node in Neo4j. The publications that mention those drug names could be linked as well (maybe via an NLP entity extraction or simply if they appear in text, which could be a future enhancement). But at least, we now have trials and papers all indexed and can be queried. - The `/search` endpoint (if exists, or internal query tool) can now surface answers that combine sources. E.g., "What do guidelines say about SGLT2 inhibitors in HFrEF?" would retrieve both trial evidence and any guideline (if we ingested guidelines too) and publications.

From an architecture perspective, these modifications remain within the existing components – we are enriching the **Ingestion & Source Adapters** part (which was Step 1 in the blueprint high-level approach [77]). All subsequent steps (normalization, chunking, etc.) accommodate the new content.

We will update any relevant **documentation** and **config** to note these new sources. For example, in our README or developer guide, list "OpenAlex, Unpaywall, Crossref" as integrated connectors, with environment variables needed (API keys for Unpaywall email, CORE API key, Semantic Scholar key, etc.). The blueprint's config can have placeholders for those secrets (like `${CORE_API_KEY}` similar to how Vault secrets were mentioned [78]).

In conclusion, the blueprint is now extended to incorporate **rich open-access literature ingestion alongside structured biomedical data**. This addendum ensures the system can gather a comprehensive knowledge base from clinical trials, drug databases, and research publications in an automated, scalable way, all unified under the multi-protocol API gateway.

Sources: The above implementation plan is informed by the official documentation and usage examples of each API and library, as cited in Sections A and B (e.g., ClinicalTrials.gov API usage [3] [79], openFDA guidelines [15], OpenAlex/pyalex usage [36], Unpaywall rate limits [16], Crossref rate limits [80], ChEMBL limits [29], etc.). These sources have guided the configuration of rate limiting, query syntax, and data mapping in our design. The integration approach follows the patterns established in the original blueprint (e.g., two-phase PDF handling [61] and adapter contract [55]), now augmented to orchestrate multiple sources seamlessly.

---

[1] [34] [53] [54] [55] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [71] [72] [73] [74] [75] [76] [77] [78] Overall_Project_Scope.md
file://file-LioogMCUPGC4LYv3GMEcc5

[2] ClinicalTrials.gov API
https://clinicaltrials.gov/data-api/api

[3] [4] [6] [7] [8] [9] [10] [56] [70] [79] ClinicalTrials.gov - BioMCP
https://biomcp.org/backend-services-reference/04-clinicaltrials-gov/

[5] API Migration Guide | ClinicalTrials.gov
https://clinicaltrials.gov/data-api/about-api/api-migration

[11] [12] [19] open.fda.gov
https://open.fda.gov/apis/

[13] [14] [15] OpenFDA - BioMCP
https://biomcp.org/tutorials/openfda-integration/

[16] REST API - Unpaywall
https://unpaywall.org/products/api

[17] openFDA API
https://simpar1471.github.io/openFDA/

[18] open.fda.gov
https://open.fda.gov/apis/try-the-api/

[20] RxNav - LHNCBC
https://lhncbc.nlm.nih.gov/RxNav/

[21] [23] [24] RxNorm API - APIs
https://lhncbc.nlm.nih.gov/RxNav/APIs/RxNormAPIs.html

[22] Using the RxNorm System
https://dcricollab.dcri.duke.edu/sites/NIHKR/KR/Using%20the%20RxNorm%20System.pdf

[25] findRxcuiByString - RxNorm API
https://lhncbc.nlm.nih.gov/RxNav/APIs/api-RxNorm.findRxcuiByString.html

[26] getDrugs - RxNorm API
https://lhncbc.nlm.nih.gov/RxNav/APIs/api-RxNorm.getDrugs.html

[27] ChEMBL - EMBL-EBI
https://www.ebi.ac.uk/chembl/

28 29 33 Integrate cheminformatics data and ClaudeAI— Part 1: ChEMBL | by Dinh Long Huynh | Medium
https://medium.com/@dinhlong240600/integrate-cheminformatics-database-and-claudeai-lets-do-it-part-1-chembl-2196382406d1

30 31 32 ChEMBL Data Web Services | ChEMBL Interface Documentation
https://chembl.gitbook.io/chembl-interface-documentation/web-services/chembl-data-web-services

35 ICD API Authentication
https://icd.who.int/docs/icd-api/API-Authentication/

36 37 38 39 GitHub - J535D165/pyalex: A Python library for OpenAlex (openalex.org)
https://github.com/J535D165/pyalex

40 41 Location object | OpenAlex technical documentation
https://docs.openalex.org/api-entities/works/work-object/location-object

42 43 Rate limits and authentication | OpenAlex technical documentation
https://docs.openalex.org/how-to-use-the-api/rate-limits-and-authentication

44 unpywall - Interfacing the Unpaywall API with Python - PyPI
https://pypi.org/project/unpywall/

45 80 Rebalancing our REST API traffic - Crossref
https://www.crossref.org/blog/rebalancing-our-rest-api-traffic/

46 47 APIs for Scholarly Resources: Get Started - Research Guides
https://guides.temple.edu/APIs

48 50 Data From Semantic Scholar Leightweight API • semanticscholar
https://kth-library.github.io/semanticscholar/

49 Semantic Scholar - APIs for Scholarly Resources - Research Guides
https://libguides.ucalgary.ca/c.php?g=732144&p=5260798

51 Searching and Evaluating Publications and Preprints Using Europe ...
https://pmc.ncbi.nlm.nih.gov/articles/PMC7614751/

52 Searching Europe PMC - EMBL-EBI
https://www.ebi.ac.uk/training/online/courses/europepmc-quick-tour/searching-and-visualising-data/