ChatGPT

Архитектурный план ETL-проекта интеграции данных из научных REST API

1. Формулировка задачи и допущения

Задача: Разработать одноразовый ETL-пайплайн на Python 3.10, который собирает данные из нескольких научных источников через их REST API, нормализует и связывает эти данные (молекулы, белковые мишени, научные публикации и идентификаторы), а затем сохраняет результаты в базе PostgreSQL для последующего аналитического использования (построение **dashboard**-отчетов, формирование «графа знаний» и т.д.). Интеграция должна охватить следующие источники: **ChEMBL**, **PubMed**, **PubChem**, **UniProt**, **Guide to Pharmacology** (IUPHAR GtoP), **OpenAlex**, **CrossRef**, **Semantic Scholar**.

Допущения и ограничения:

- Пайплайн выполняется *однократно* в режиме bulk-выгрузки (единовременная сборка актуальных данных). Регулярные периодические обновления не требуются, хотя архитектура должна допускать потенциальное обновление или доработку в будущем.
- **Только REST API:** подключение к источникам данных осуществляется исключительно через их официальные REST API. Предоставляемые источниками дампы или базы данных не используются (например, локальный дамп ChEMBL исключен — только веб-сервисы).
- Используемый стек: Python 3.10 с разрешением на стандартные библиотеки и популярные open-source библиотеки: HTTP-клиенты (`requests` или `httpx` для вызовов API), схемы данных (`pydantic` для валидации/нормализации), ORM/SQL-библиотека (`sqlalchemy` для взаимодействия с PostgreSQL), а также утилиты concurrency (например, `asyncio` при необходимости) и другие вспомогательные библиотеки, не противоречащие ограничениям.
- **PostgreSQL** выступает конечным хранилищем интегрированных данных. Предполагается наличие доступа к БД и возможность создания необходимых схем/таблиц.
- Цель интеграции — подготовить унифицированную витрину данных для аналитики. Данные из разных источников должны быть **нормализованы** к общему представлению и логически связаны (например, объединение записей об одной молекуле из разных БД, связь молекул с их биологическими мишенями, связь публикаций с соответствующими объектами).
- **Naming policy:** Стиль кода и структура проекта будут организованы в соответствии с принятыми в проекте `bioetl` соглашениями о наименованиях и разбиении на модули. Это значит, что будем придерживаться строгих правил оформления: классы — в PascalCase (например, `ChemblClient`), функции — в snake_case (`load_chembl_activities`), константы — `UPPER_SNAKE_CASE` и т.д., а структура каталогов повторит шаблон `src/bioetl/...` с разделением на слои (clients, pipelines, schemas, core и пр.) согласно документу *11-naming-policy.md*. Такая дисциплина обеспечит единообразие и облегчит поддержку проекта.

2. Характеристики источников данных

Перед началом разработки определим особенности каждого внешнего источника — его назначение, используемые REST-эндпоинты, требования по аутентификации (API-ключи) и лимиты запросов. Ниже приведен обзор по каждому сервису:

- **ChEMBL** – база данных биологически активных химических соединений, особенно лекарственных веществ, с информацией о их *биоактивности* и мишенях. *Цель интеграции:* получить список химических веществ (молекул) и их взаимодействия с биологическими целями (таргетами), а также внешние идентификаторы этих веществ.
- **API и эндпоинты:** ChEMBL предоставляет REST API (базовый URL: `https://www.ebi.ac.uk/chembl/api/data/`). Основные сущности: `molecule` (информация о соединении по его ChEMBL ID), `target` (информация о мишени по ID), `activity` (биоактивности, связывающие молекулы и мишени). Например, `GET /chembl/api/data/molecule/CHEMBL25.json` вернет JSON с данными по молекуле **CHEMBL25**. Также доступен эндпоинт поиска или листинга: `GET /chembl/api/data/molecule?limit={N}&offset={M}` для постраничного перебора молекул (позволяет пройтись по всем записям, задавая постранично).
- **Ключи и доступ:** API ChEMBL открыт, API-ключ по умолчанию не требуется. Однако, для большого объема запросов EBI может предоставлять *API key* для повышения лимитов. Без ключа на публичном API действует ограничение: **не более ~1 запроса в секунду** [1]. Это важно учитывать при bulk-выгрузке: прямое получение миллионов записей последовательно по 1/сек будет чрезвычайно долгим. Поэтому либо потребуется запрашивать ключ (увеличивая throughput), либо реализовать параллелизацию с учетом лимита на одно соединение (например, 1 req/sec с нескольких IP, что малореально, или получить ключ через регистрацию).

- **Дополнительно:** Формат ответа – JSON (по умолчанию ChEMBL API может возвращать JSON или XML; мы явно используем `.json` суффикс). ChEMBL содержит различные внешние ссылки внутри своих записей, напр. для мишеней часто указывается унифицированный код UniProt (accession), для соединений – InChIKey, синонимы, ссылки на PubChem и др., что поможет в последующей стыковке данных.

- **PubMed** – крупнейшая библиографическая база научных публикаций (биомедицинские статьи). *Цель интеграции:* получить сведения о публикациях (название, авторы, журнал, год) по идентификаторам статей, связанным с нашими молекулами/мишенями. В частности, ссылки на PubMed могут встречаться в других базах (ChEMBL, Guide to Pharmacology и др.) как доказательства взаимодействий.

- **API и эндпоинты:** PubMed предоставляет REST-интерфейс через NCBI E-utilities. Например, для получения подробной информации по PMID (идентификатор PubMed) используется `EFetch` или `ESummary`: `GET https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esummary.fcgi?db=pubmed&id=<PMID>&retmode=json` возвращает JSON с полями статьи (или `efetch.fcgi?db=pubmed&id=<PMID>&rettype=XML` – XML с полным описанием). Можно также искать статьи по ключевым словам (`esearch.fcgi`) или получать связанные ID списком. В нашем случае, скорее всего известны конкретные **PMID** из других источников, и мы будем делать прямой вызов по ним.

- **Ключи и лимиты:** NCBI E-utilities бесплатны, но требуют указания контактного email в запросах &tool, &email для интенсивного использования. Без API-ключа действует ограничение ~**3 запроса в секунду** с IP; при использовании личного API Key (получается

на сайте NCBI) лимит повышается до ~10 запросов/сек. Так как наш сценарий – bulk-запросы, рекомендуется либо ограничиться частотой ~3 RPS, либо регистрировать ключ разработчика, чтобы ускорить извлечение многих статей. Количество статей, связанных с одной молекулой/мишенью, обычно относительно невелико, поэтому проблем с лимитом быть не должно, но нужно учитывать общее число обращений (например, если выгружается тысячи статей, нужен throttle). Формат выдачи – JSON или XML, будем использовать JSON для удобства парсинга.

- **PubChem** – открытая база химических веществ (NIH), дополняющая ChEMBL. *Цель интеграции:* обогащение информации о молекулах (например, химическая структура, молекулярная формула, синонимы, идентификаторы **CID**, **InChIKey**, возможно ссылки на безопасность и пр.). Также PubChem поможет сопоставить соединения между базами (через InChIKey или CID).

- **API и эндпоинты:** PubChem предоставляет REST API, известный как **PUG-REST**. Он позволяет получать данные о соединениях по разным идентификаторам. В нашем случае:
  - Если имеем InChIKey или название соединения, можно использовать эндпоинт поиска, например: `GET https://pubchem.ncbi.nlm.nih.gov/rest/pug/compound/inchikey/<InChIKey>/JSON` для получения CID по InChIKey, или `compound/name/<Name>` и т.п.
  - Имея **CID** (PubChem Compound ID), можно запросить подробную информацию: `GET /rest/pug/compound/cid/<CID>/JSON?record_type=full` (возвращает подробное описание вещества, включая свойства, синонимы, ссылки на другие DB).
  - Можно запрашивать списки сразу по нескольким CID (Batch), ограничивая поля через параметры. Например, `GET compound/cid/1,2,3/property/MolecularFormula,MolecularWeight/JSON`.
- **Ключи и лимиты:** PubChem не требует API-ключа, сервис общедоступный. Однако, есть *правило вежливости*: рекомендовано не более **5 запросов в секунду** (максимум ~400 запросов в минуту) [2]. Кроме того, один запрос может возвращать до 10 000 записей за раз (по параметру `limit`), что позволяет пачками получать данные, снижая нагрузку на сетевые соединения [3]. Мы будем укладываться в эти рамки: при массовом запросе свойств лучше объединять ID и запрашивать их оптом, чем стучаться по одному (уменьшит общее число HTTP-вызовов).

- **Примечания:** PubChem содержит пересечения с ChEMBL (многие хим. соединения присутствуют в обеих БД). Мы будем использовать *InChIKey* как унифицированный идентификатор молекулы: ChEMBL предоставляет InChIKey для каждого соединения, по нему мы сможем быстро найти соответствующий **CID** в PubChem и дополнительные данные. Это часть стратегии нормализации.

- **UniProt** – универсальная база данных белков (белковые последовательности, генные названия, функции). *Цель:* получить стандартизированную информацию о **таргетах** – белках-мишенях, с которыми взаимодействуют наши молекулы. Другие источники (ChEMBL, GtoP) дают ссылки на UniProt (через accession ID), поэтому мы сможем получить официальное название белка, ген, организм, и другие аннотации для построения единого справочника мишеней.

- **API и эндпоинты:** Актуальный REST API UniProt (2025 г.) – это сервис на `https://rest.uniprot.org`. Для получения записи белка по accession (например, `P12345`) используется: `GET https://rest.uniprot.org/uniprotkb/P12345.json` (вернет JSON с полями UniProtKB). Также есть поиск: `GET /uniprotkb/search?`

`query=<term>&format=json&size=500` (например, можно искать по gene символу или названию). UniProt предоставляет возможность батчевого запроса нескольких ID через *ID Mapping API*, но в нашем случае, если у нас есть все accession, можно получать их поштучно или запросами по 500 (макс размер страницы).

- **Ключи и лимиты:** API открыт, ключ не требуется. Тем не менее, UniProt накладывает ограничения на интенсивность: одновременных параллельных запросов к их API лучше делать немного (их документация рекомендует не запускать множество потоков одновременно). Мы планируем обращаться постранично (500 записей за запрос) или последовательно с небольшой параллелизацией (например, 2-3 одновременных вызова максимум) для ускорения. Существенным ограничением является объем данных: полный UniProtKB огромен, но нам нужен лишь ограниченный набор записей (мишени, фигурирующие в ChEMBL/GtoP). Ожидается, что таких уникальных белков будет, скажем, сотни или тысячи, что вполне допустимо вычитать по API. Формат – JSON, который мы распарсим Pydantic-схемами.

- **Guide to Pharmacology (IUPHAR/BPS GtoP)** – экспертно курируемая база данных фармакологических мишеней и их лигандов (лиганд = активное соединение: лекарство, токсин и т.д.). *Цель:* извлечь подтвержденные пары «лиганд-таргет» с качественными аннотациями и литературными ссылками. GtoP предоставит относительно компактное ядро знаний: список **лигандов** (молекул, часто зарегистрированных лекарств), список **таргетов** (рецепторы, ферменты и пр.), а главное – **интеракции** между ними с указанием типа действия и ссылок на публикации.

- **API и эндпоинты:** GtoP имеет REST web services (база URL: `https://www.guidetopharmacology.org/services/`, возвращающие JSON). Основные вызовы:
  - `GET /ligands` – список всех лигандов (можно фильтровать по названию, типу и т.д.). Каждый лиганд имеет свой `ligandId`.
  - `GET /ligands/{ligandId}` – подробная информация о лиганде (наименование, синонимы, SMILES, внешние ссылки: часто указываются ChEMBL ID, PubChem CID, UniProt для пептидов, etc).
  - `GET /targets` – список мишеней (с возможностью фильтрации по классу, например GPCR, Ion Channel и т.д., или поиску по имени/гену).
  - `GET /targets/{targetId}` – детали по конкретной мишени (название, ген, семейство, внешний UniProt ID и пр.).
  - **Самое важное:** `GET /interactions` – список всех пар «таргет–лиганд» в базе. Можно фильтровать по ligandId или targetId для отдельных выборок, либо без параметров получить полный список взаимодействий. Запрос всех взаимодействий может вернуть несколько тысяч записей (документация упоминает, что получение тысячи+ результатов может занять несколько секунд). Каждая запись содержит идентификатор лиганда, идентификатор мишени, тип взаимодействия (например, агонист, ингибитор) и часто **references** – ссылки на научные статьи (идентифицируемые по PMID или DOI).
  - `GET /references` – предоставляет информацию о литературных источниках по их internal ID или весь набор ссылок. Однако удобнее сразу из взаимодействия извлечь PMID/DOI, а детали о статье уже получить через PubMed/CrossRef.
- **Ключи и лимиты:** API GtoP не требует ключа, свободен для использования. Явных жестких ограничений по RPS нет (в разумных пределах). Возможно, есть базовые ограничения на уровне веб-сервера (например, < 10 одновременных запросов), но в рамках нашего сценария это не проблема. Мы можем извлечь все лиганды, все таргеты и все взаимодействия по 1 запросу каждый (если размер ответа умеренный). Если ответы

слишком большие, при необходимости задействуем фильтрацию/постраничный запрос (напрямую документация о пагинации не упоминается, возможно, все отдается списком).

- **Примечания:** Данные GtoP крайне полезны для сшивки: каждый лиганд обычно имеет поле `chemblId` (если вещество есть в ChEMBL) или `pubChemCID`, то есть можно сразу связать объекты GtoP с соответствующими из других баз. У таргетов GtoP, как правило, указаны UniProt accession для каждого белка. Таким образом GtoP будет отправной точкой интеграции, предоставляя ключи для связи между системами.

- **OpenAlex** – открытый индекс академических публикаций (альтернативный открытый аналог Microsoft Academic Graph). *Цель:* получить метаданные публикаций (например, название, список авторов, DOI, кол-во цитирований, темы) по известным идентификаторам. OpenAlex может дополнить или заменить запросы к PubMed/NCBI, предоставив, например, данные по DOI статьи.

- **API и эндпоинты:** Базовый URL API: `https://api.openalex.org`. Основной ресурс для нас – **works** (научные работы). Эндпоинты:
  - `GET /works/{openalex_id}` или `/works/https://doi.org/<DOI>` – получить запись статьи по OpenAlex ID или DOI (OpenAlex умеет принимать DOI URI прямо). Например, `GET /works/https://doi.org/10.1038/s41586-020-2649-2` вернет JSON по статье с данным DOI.
  - `GET /works?filter=doi:<DOI>` – альтернатива для DOI.
  - Также можно искать по названию или автору, но нам, вероятно, достаточно прямого обращения по DOI, который будет известен из CrossRef или GtoP.
- **Ключи и лимиты:** OpenAlex API публичный и **не требует ключа**, но имеет ограничения: **не более 100 000 запросов в день и 10 запросов в секунду** [4] [5]. Этого лимита вполне достаточно для наших целей (число статей явно меньше 100k). Для лучшей производительности OpenAlex рекомендует указывать свой email в параметре `mailto` во всех запросах (чтобы идентифицировать запросы и получать приоритет в «polite pool»). Мы будем это делать, особенно если объём обращений значительный.

- **Примечания:** OpenAlex предоставляет богатый JSON: помимо базовых полей (title, authors, venue, year) есть аналитика (количество цитирований, список ссылок на другие работы и пр.). В контексте нашего проекта, возможно, слишком детально это не нужно — достаточно базовых библиографических данных. Тем не менее, OpenAlex может пригодиться для валидации: например, если CrossRef не найдёт DOI или мы хотим сравнить данные.

- **CrossRef** – основной регистрационный агент DOI, предоставляющий метаданные по DOI научных публикаций. *Цель:* по известному DOI получать атрибуты публикации (название, авторы, журнал, год) и, при необходимости, список ссылок (reference list) или другие детали. CrossRef можно использовать как основной источник информации о статье по DOI, дополняя PubMed (который оперирует PMID).

- **API и эндпоинты:** REST API CrossRef доступен по `https://api.crossref.org/v1/works/` (или `/v1/works/DOI`). Например, `GET /v1/works/10.1038/s41586-020-2649-2` возвращает JSON с метаданными по этому DOI. Также можно искать: `GET /v1/works?query=<text>` – полнотекстовый поиск, но он может быть шумным. Предпочтительно, если у нас есть DOI или PMID, лучше конвертировать PMID в DOI (через eutils или S.Scholar) и идти напрямую. CrossRef обычно возвращает: title, authors, issued date, container (journal) и пр.

- **Ключи и лимиты:** API открыт, ключ или регистрация не требуются. Однако CrossRef имеет *polite limit*: **до 50 запросов в секунду с IP** [6] [7]. Это очень щедро; вероятно, мы вообще не достигнем такого количества (публикаций у нас не десятки тысяч). В любом случае, рекомендуется в User-Agent или параметре указывать контактный email, чтобы показывать добросовестность. Мы можем безопасно слать запросы с частотой ~5-10 RPS без опасений.

- **Примечания:** CrossRef в ряде случаев может не содержать информации о некоторых старых статьях или малоизвестных журналах. Тогда выручит PubMed или Semantic Scholar. Но по DOI, как правило, CrossRef – самый прямой путь получить библиографию.

- **Semantic Scholar** – AI-поисковик и база научных публикаций с дополнительной аналитикой (цитаты, влияние). *Цель:* резервный или дополнительный источник данных о публикациях. Он может дать, например, список цитируемой литературы по PMID/DOI, или помочь преобразовать PMID→DOI. Также S. Scholar предоставляет оценки влияния, которые могут быть интересны, хотя для нашей задачи (витрина хим. знаний) это вторично.

- **API и эндпоинты:** Необходимо отметить, что S.Scholar предоставляет API по ключу. Существуют два варианта: старый `REST` и новый Graph API. Мы будем опираться на актуальный **Semantic Scholar API v1** (Graph). Эндпоинт для работы: `GET https://api.semanticscholar.org/graph/v1/paper/<PaperID or DOI>?fields=<...>` где в `fields` перечисляются нужные поля (например, title, authors, year, citationCount, externalIds). Можно использовать DOI напрямую вместо внутреннего ID. Пример: `GET /graph/v1/paper/DOI:10.1038/s41586-020-2649-2?fields=title,year,authors`.
- **Ключи и лимиты:** Для публичного использования нужен **API Key**, выдаваемый по запросу (бесплатно с базовыми ограничениями). Без ключа доступ ограничен (незарегистрированное использование через веб – до 100 запросов/5 минут по некоторым данным [8] [9], что неудобно). С базовым API-ключом лимит ~**1 запрос/сек** (60 в минуту) [10], с возможностью увеличения по согласованию. То есть S.Scholar – самый «тяжелый» по ограничениям сервис из всех перечисленных. Для нашего bulk-процесса придется либо получить ключ и учитывать этот лимит, либо ограничить использование этого API.
- **Тактика использования:** Semantic Scholar будем задействовать опционально на финальных этапах интеграции, например, чтобы дополнить данные, которые не удалось взять из CrossRef/PubMed. Например, он может конвертировать **PMID→DOI** (в поле `externalIds` выдаются PMID, DOI, MAG и т.д.), или предоставить счетчики цитирований (`citationCount`) для расширенной аналитики. Но на критическом пути (получение названий статей и основных метаданных) мы можем обходиться PubMed+CrossRef, чтобы не упереться в жёсткие лимиты S.Scholar.

Вывод по источникам: Каждый источник играет определенную роль в нашем **графе знаний**. ChEMBL и GtoPdb дают исходные списки молекул и их связей с мишенями, UniProt нормализует информацию о самих мишенях, PubChem нормализует информацию о самих молекулах, а PubMed/CrossRef/OpenAlex/SemanticScholar дают информацию о публикациях, подтверждающих эти связи. Все API возвращают данные в JSON (либо имеют опцию JSON), что упрощает их обработку. Важно учитывать ограничения по скорости: наиболее строгие у ChEMBL (1 RPS) и Semantic Scholar (1 RPS с ключом), умеренные у PubMed (~3 RPS) и PubChem (~5 RPS), щедрые у CrossRef (50 RPS) и OpenAlex (10 RPS). Поэтому при проектировании нужно распределить вызовы и при необходимости реализовать очереди или задержки, чтобы уложиться в лимиты, особенно для ChEMBL и S.Scholar.

3. Архитектура Python-проекта (слои, модули, структура каталогов, классы)

Проект будет организован многослойно, с четким разделением ответственности каждого компонента. Основные уровни архитектуры:

- **Уровень клиентов API (external clients)** – классы/модули, отвечающие за подключения к внешним сервисам и получение сырых данных.
- **Уровень схем и моделей (schemas/models)** – классы данных (Pydantic-модели) для унифицированного представления основных сущностей (молекула, мишень, публикация и т.д.), а также для некоторых «сырых» структур если нужно.
- **Уровень ETL-пайплайнов (pipelines)** – набор модулей, реализующих конкретные этапы ETL (extract, transform, load) для каждой комбинации источник/сущность. Здесь происходит orchestration: вызов клиентов, преобразование данных к схеме и передача в хранилище.
- **Уровень ядра (core)** – общая логика и утилиты: базовые классы, вспомогательные функции (например, функции нормализации, конвертации форматов), механизмы оркестрации (например, класс PipelineRunner), конфигурирование и т.п.
- **Уровень хранения (storage)** – модуль для работы с базой данных: описание ORM моделей (SQLAlchemy), создание подключения, методы записи/чтения.
- **Вспомогательный уровень**: CLI и утилиты – отдельные модули/скрипты для запуска пайплайна, парсинга аргументов командной строки, а также общие утилиты (например, rate limiter, логирование, retry-логика для запросов).

Согласно политике `bioetl`, структура каталогов внутри репозитория будет следующей (корневой пакет назовем `bioetl`):

```
src/bioetl/
    api/              # Публичный программный API (входные точки, фасады) — возможно, не критично для однократного запуска, но может содержать high-level функции запуска.
    cli/              # CLI-скрипты для запуска ETL из командной строки.
    clients/          # Клиенты для внешних сервисов (REST API).
    core/             # Ядро: общие компоненты ETL (базовые классы, конфиги, оркестраторы).
    pipelines/        # Пайплайны ETL, организованные по схеме <провайдер>/<сущность>/<этап>.
    schemas/          # Схемы данных (Pydantic-модели) и их валидаторы/конвертеры.
    utils/            # Утилиты (не зависят от конкретного провайдера или пайплайна).
    ... (tools/, configs/, tests/, docs/ и пр. согласно соглашению)
```

Разберем основные компоненты и классы:

- **Модуль** `src/bioetl/clients/`: здесь будут реализованы классы-клиенты для каждого внешнего API. Например:
- `ChemblClient` – класс для работы с ChEMBL API. Внутри: базовый URL, методы `fetch_molecule(chembl_id)`, `fetch_target(target_id)`, `fetch_activities(filter)` и пр. Он инкапсулирует вызовы `requests` / `httpx`,

обработку ответа, повтор попытки при ошибке, возможное логирование. Можно реализовать общий базовый класс `BaseClient` с общими методами (например, сессия httpx, метод `_get(url, params)` с обработкой исключений и задержкой), от которого наследовать конкретных клиентов (ChemblClient, PubmedClient, ...).

- `PubmedClient` – для запросов к NCBI (можно обернуть E-utilities вызовы, формируя URL).
- `PubchemClient` – для вызовов PUG-REST (методы например `get_compound_by_inchikey(key)` или `get_compounds_by_cid([...])`).
- `UniprotClient` – обертки над запросами к UniProt (метод `get_protein(accession)`).
- `IupharClient` (или `GuidetopharmaClient`) – для GtoPdb (методы `get_all_ligands()`, `get_all_targets()`, `get_interactions()`).
- `OpenAlexClient`, `CrossrefClient`, `SemanticScholarClient` – для соответствующих сервисов (например, `get_work_by_doi(doi)`).
- Каждый клиент будет использовать либо `requests` (в синхронном режиме, выполняя вызовы последовательно или из потоков), либо `httpx.AsyncClient` (позволяя запустить несколько запросов одновременно, особенно к разным сервисам). Выбор стратегии может быть задан конфигурационно. Например, для ChEMBL стоит явно встроить задержку 1 сек между запросами (если нет ключа), для PubChem можно разрешить 5 параллельных. Возможно, придется встроить простейший **rate limiter** (в `utils/`), который клиенты будут вызывать перед отправкой запроса.
- В конструкторе клиента можно принимать параметры (базовый URL на случай изменения, API-ключи если требуются, таймауты). Например, `SemanticScholarClient(api_key=...)` будет вставлять ключ в заголовки Authorization.

- **Важно:** Клиенты возвращают данные в том виде, как есть (сырые JSON или отфильтрованные dict). Они не должны принимать решений по структуре БД, это задача уровней выше. Их цель – надежно получить и десериализовать ответ. Для удобства, можно сразу привязывать к Pydantic-моделям (см. следующий пункт): например, `ChemblClient.fetch_molecule` может сразу возвращать объект `ChemblMoleculeSchema` (Pydantic), если мы определим модель, отражающую поля API. Либо возвращать dict, а преобразование делать отдельно. Мы выберем вариант с минимальной логикой в клиентах: получение JSON -> возврат dict/Pydantic, но без слишком сложной трансформации.

- **Модуль** `src/bioetl/schemas/`: здесь определяются схемы данных (модели) для унифицированных сущностей. Согласно naming conventions, классы Pydantic должны оканчиваться на `Schema` или `Model`.

- `MoleculeSchema` – Pydantic-модель, описывающая сущность «Молекула» в нашем интегрированном виде. Поля, например: `chembl_id` (str, optional), `pubchem_cid` (int, optional), `iuphar_id` (int, optional), `name` (str), `inchikey` (str), `smiles` (str, optional), `drug_type` (str, optional, тип вещества – если есть, напр. синтетический/биологический), `synonyms` (List[str]), и т.д. Цель – объединить в одном объекте все ключевые атрибуты молекулы из разных источников. Pydantic позволит валидировать типы (например, что CID это int, SMILES строка, длина InChIKey 27 символов и т.п.) и при необходимости выполнять конверсию (например, привести название к единому регистру или нормализовать формат).
- `TargetSchema` – модель для сущности «Таргет» (белок/биомишень). Поля: `uniprot_id` (str, основной идентификатор), `name` (str, название мишени), `gene` (str, ген), `organism` (str), `chembl_id` (str, если есть Chembl Target ID), `iuphar_id` (int, если есть), возможно классификации (класс рецепторов и т.п.). Эта схема аккумулирует информацию из UniProt (имя, организм), GtoP (ID, тип мишени) и ChEMBL (target_chembl_id).

- `PublicationSchema` – модель публикации. Базовые поля: `doi` (str, уникальный, если есть), `pubmed_id` (int, может быть None если нет), `title` (str), `journal` (str), `year` (int), `authors` (List[str] или строка со списком авторов), `citation_count` (Optional[int] – если берем из S.Scholar/OpenAlex), и `url` (формируем, например, из DOI). Эту модель мы будем наполнять либо из CrossRef/OpenAlex (они дают почти все перечисленное), либо из PubMed (title, etc).
- **Raw schemas:** Можно также завести схемы, отражающие формат сырых ответов, например `ChemblMoleculeSchemaRaw` под JSON от ChEMBL. Однако это может дублировать описание, поэтому, возможно, мы обойдемся без этого: будем сразу маппить нужные поля в наши унифицированные схемы. Тем не менее, Pydantic можно использовать и для парсинга сложных вложенных JSON (например, ответ ChEMBL `molecule` содержит вложенные списки биоактивностей? Но в основном, наверное, нет, они отдельным запросом).
- **Связи в моделях:** Pydantic модели можно вкладывать друг в друга. Но здесь главные сущности будут связаны через внешние ключи в БД, а не вложены. Т.е. `MoleculeSchema` не будет содержать напрямую список Target'ов – связи будем хранить отдельно. Однако можем завести отдельную модель для взаимодействия: `InteractionSchema` с полями `molecule_chembl_id`, `target_uniprot_id`, `activity_type` (например, inhibition/agonism), `reference_ids` (список PubMed ID или DOI подтверждений). Это по сути запись отношения, которая потом ляжет в связь в БД.
- **SQLAlchemy ORM vs Pydantic:** Эти модели используются для валидации и как промежуточные контейнеры. Параллельно, опишем ORM-модели (в `storage` уровне) для таблиц. Их поля будут соответствовать Pydantic-схемам, и мы настроим преобразование (например, PydanticModel.dict() -> ORM instance). Можно также воспользоваться интеграцией Pydantic + SQLAlchemy (Pydantic v2 имеет Basemodel.from_orm etc), но можно и вручную.

- **Именование:** Соблюдаем PascalCase + Suffix, как сказано: `MoleculeSchema`, `PublicationModel` и т.д., согласно соглашению [11] [12].

- **Модуль** `src/bioetl/pipelines/`: здесь расположены пакеты пайплайнов по источникам и сущностям, разделенные на этапы. Структура каталогов будет соответствовать шаблону `pipelines/<provider>/<entity>/<stage>.py` [13] [14]:

- *Пример организации:*
  - `pipelines/chembl/compound/extract.py` – модуль, который знает как извлечь (extract) данные о соединениях из ChEMBL.
  - `pipelines/chembl/compound/transform.py` – модуль для преобразования (transform) этих сырых данных ChEMBL в наши унифицированные `MoleculeSchema`.
  - `pipelines/chembl/compound/load.py` (или `write.py` по терминологии) – модуль, записывающий получившиеся объекты в БД (таблицу molecules).
  - Аналогично: `pipelines/chembl/target/extract.py` / `transform.py` / `load.py` для таргетов ChEMBL (если нужны, ChEMBL имеет target info).
  - `pipelines/chembl/activity/extract.py` – сбор биоактивностей (связей) из ChEMBL (например, `GET /activity?molecule_chembl_id=...`).
  - `pipelines/iuphar/ligand/extract.py`, `transform.py` и т.п. для лигандов GtoP; `iuphar/target/*` и `iuphar/interaction/*`.
  - `pipelines/uniprot/protein/extract.py` (получение инфо по белкам), `transform.py` (в `TargetSchema`), `load.py`.

- `pipelines/pubchem/compound/extract.py` (доп. свойства молекул), `transform.py`, `load.py` (возможно, load сливается с chembl's load, т.к. это дополнил поля).
- `pipelines/pubmed/publication/extract.py`, `transform.py`, `load.py`.
- `pipelines/openalex/publication/extract.py` ... и т.д.
- Потенциально, `pipelines/semanticscholar/publication/extract.py` если будем делать отдельный этап.
- **Этап** `run` **или orchestration:** Можно в каждом `<entity>` подпакете сделать файл `run.py`, который по порядку вызывает extract->transform->load для данной сущности в контексте данного провайдера. Или один уровень выше – `pipelines/chembl/compound/run.py` orchestrates chembl compound pipeline, и `pipelines/chembl/run.py` orchestrates all chembl pipelines? По вкусу. Согласно соглашению, есть понятие stage `run` как high-level orchestration [15]. Вероятно, стоит делать `run.py` для каждого `<provider>/<entity>` как точку входа этого мини-пайплайна, а еще один общий orchestrator.

• **Содержимое этапов:**

- `extract.py`: содержит функцию или класс, которая с помощью соответствующего клиента API получает данные. Например, в `chembl/compound/extract.py` функция `extract_all_chembl_compounds(client: ChemblClient) -> List[dict]` выполняет серию вызовов `client.fetch_molecule` или использует клиент для выгрузки постранично всех молекул по критерию (например, все молекулы, которые являются лекарствами – ChEMBL позволяет фильтровать по флагу molecule_type=Small molecule, max_phase>0). Возможно, есть смысл ограничить перечень молекул (например, только те, что упомянуты в GtoP, чтобы не выгружать весь ChEMBL). Эти решения могут быть зашиты в конфигурацию.
- `transform.py`: функция `transform_compounds(raw_records: List[dict]) -> List[MoleculeSchema]` – проходит по сырым записям, создавая объекты `MoleculeSchema`. Здесь выполняется нормализация: приводим поля к нужным типам, заполняем недостающие значения, отфильтровываем лишнее. Например, если ChEMBL-ответ дал `pref_name` (имя вещества) и `molecule_structures.smiles`, мы заносим это в `name` и `smiles`. Если `molecule_structures.standard_inchi_key` – кладем в `inchikey`. Если есть `molecule_properties.full_mwt` – можем сохранить мол. вес, если нужно для аналитики. Аналогично для других источников: transform берет output extract-а и выдает валидированные Pydantic объекты нашей схемы.
- `load.py`: функция `load_compounds(records: List[MoleculeSchema], db: Session)` – сохраняет список молекул в базу. Здесь используем SQLAlchemy-сессию. Каждый `MoleculeSchema` преобразуется в ORM-модель `Molecule` (например, через конструктор: `mol = Molecule(**schema.dict())`) и добавляется в сессию. Можно оптимизировать через `session.bulk_insert_mappings` или `session.bulk_save_objects` для больших списков. После добавления всех – коммит транзакции. Если нужно избежать дубликатов: прежде чем вставлять, можно проверять по уникальному ключу (например, по InChIKey или chembl_id) нет ли уже такой записи. Но в нашем случае мы сначала очистим или создадим таблицы пустыми (fresh load), т.к. это initial bulk load.
- **Оркестрация:** Если выбираем подход с `run.py` в pipelines, то, например, `pipelines/chembl/compound/run.py` будет делать: `raw = extract.extract_all_chembl_compounds(ChemblClient(...)); normalized =`

`transform.transform_compounds(raw); load.load_compounds(normalized, session)`. То есть три шага подряд. Похожий run будет для `chembl/target`, `iuphar/ligand` и т.д.

- Кроме того, некоторые пайплайны зависят друг от друга: например, чтобы связать молекулу с таргетом, надо сначала иметь загруженные молекулу и таргет. Вероятно, сначала грузим все справочники (молекулы, таргеты, публикации), затем связи. Поэтому глобально нужна последовательность:
- Выгрузить **все молекулы** (ChEMBL + GtoP + PubChem интегрировано) -> загрузить в таблицу molecules.
- Выгрузить **всех таргетов** (UniProt + GtoP + ChEMBL) -> загрузить в таблицу targets.
- Выгрузить **связи** молекула-таргет (из ChEMBL activities и GtoP interactions) -> маппинг на ID молекул/таргетов -> загрузить в таблицу связей.
- Собрать **список публикаций**, необходимых для подтверждения (собрать все PMID/DOI из шагов 3, а также, возможно, из свойств молекул/таргетов если там указаны reference). Выгрузить по ним данные из PubMed/CrossRef/OpenAlex -> загрузить в таблицу publications.
- Загрузить **связи публикаций**: вероятно, связь «публикация описывает взаимодействие молекулы и таргета». Можно иметь отдельную таблицу `interaction_references` (связь многие-ко-многим: связка на запись связи и на публикацию). Либо поле в таблице связей, указывающее reference (если предполагается только одна основная ссылка, но лучше учитывать множественные).
- Также могут быть связи «публикация про молекулу» напрямую или «про таргет», но это, возможно, излишне; мы фокусируемся на связях подтверждения взаимодействий.
- Каждый из этих шагов можно реализовать совокупностью pipeline модулей. Например, GtoP interactions дадут сразу молекула-таргет-публикация (PMID) тройки; ChEMBL activities тоже могут иметь reference (но часто вместо PMID ChEMBL указывает DOI или Journal reference, это нюанс).
- **Поток данных:** ETL-архитектура позволит либо собирать все в память, либо стримить пачками. Предпочтительно, если объемы большие, делать пакетную обработку: например, не вытаскивать сразу 2 миллиона молекул в лист, а читать по страницам и сразу загружать частями. Это может быть реализовано в `extract` (итератор по страницам) и в `load` (коммит каждые N записей). Однако для упрощения можно сперва реализовать «всё в память» если ожидаем разумные объемы (например, GtoP ~ 1400 лигандов, это ок; ChEMBL however >2 млн молекул – явно не ок для памяти и времени). Поэтому, вероятно, *ограничим набор* молекул, иначе ChEMBL полный недостижим через API. Ограничение может быть: берем молекулы до определенной максимальной фазы (только препараты в клиническом использовании, их ~2000-10000, это выполнимо). Либо ориентируемся на пересечение с GtoP (все лиганды GtoP наверняка в ChEMBL, их ~<=3000, это уже manageable).
- **Классы vs функции:** Pipeline-этапы можно оформить функциями (в модуле extract.py функция extract_xxx). Либо сделать класс `ChemblCompoundPipeline` с методами `.extract()`, `.transform()`, `.load()`. В духе `bioetl` логично держать их как модуль с функциями или с единственной точкой входа. В naming policy явного требования нет, но тесты ожидают `test_extract.py` etc. Вероятно, функции — нормально.
- **Логирование и отладка:** В каждом этапе, особенно extract, стоит логировать сколько записей получено, время вызова, URL. В transform — возможно, логировать пропуски/исправления (например, «10 записей пропущено из-за отсутствия

обязательных полей»). В load — время вставки, количество вставленных строк. Эти логи помогут в QA и оценке производительности.

- **Модуль** `src/bioetl/core/`: это сердце системы, содержащее общие механизмы:

- `core/pipeline_runner.py` – класс **PipelineRunner** (PascalCase имя) или функция, которая знает последовательность запуска всего ETL. Например, метод `run_all()` вызывает по порядку нужные пайплайны, или позволяет выбрать определенный subset. Можно настроить через конфигурацию YAML какие пайплайны активировать.
- `core/base_pipeline.py` – абстрактный класс **BasePipeline** с общими атрибутами (например, имя провайдера, сущность, конфиг, логгер) и, возможно, абстрактным методом `run()` или конкретными `.extract/.transform/.load` если решили классом. Но это не строго обязательно, функции тоже подойдут.
- `core/database.py` – тут инициализация подключения к PostgreSQL (создание `engine` SQLAlchemy, `SessionLocal` sessionmaker, функция для Base.metadata.create_all). Также описание ORM-моделей. Можно выделить `models.py`:
  - Класс `Molecule` (наследник `Base` декларативного SQLAlchemy) с полями: `id` (PrimaryKey), `chembl_id` (nullable=True), `pubchem_cid`, `inchikey`, `name`, ...; уникальный индекс на `inchikey` или `chembl_id` чтобы не было дублей; возможно, связь (relationship) на таблицу связей.
  - Класс `Target` (id, uniprot_id, gene, name, organism, ... + alt ids).
  - Класс `Publication` (id, doi, pmid, title, etc).
  - Класс `Interaction` (id, molecule_id -> FK Molecule, target_id -> FK Target, type, source_db, etc). Если нужны многие-ко-многим references, можно сделать:
  - Класс `InteractionReference` (interaction_id, publication_id, primary key композитный).
  - Или упростить: если храним только одну основную публикацию, достаточно поля publication_id в Interaction (но лучше many-to-many, т.к. разные источники могут дать несколько PMID).
  - Таблицы связей можно нормализовать до 3NF, но здесь достаточно понятных foreign keys с index'ами для быстрых join-ов (для дашбордов будут нужны join molecule->interaction->target, etc).
- `core/config.py` – если планируется конфигурация (например, YAML с перечислением провайдеров, credentials, опций pipelines). Можно подключить чтение `configs/providers.yaml` где перечислены provider: e.g. chembl, pubchem, etc (как в naming rules упоминалось) [16]. Также можно настроить `configs/pipelines/chembl_compound.yaml` с параметрами (например, `max_phase: 4`, `min_activity_count: 1` для фильтра), но это опционально. Основные константы (базовые URL, default limit per page, etc) можно либо захардкодить, либо вынести в config.
- `core/normalization.py` – здесь можно хранить функции общего характера нормализации: например, `_normalize_smiles(smiles: str) -> str` (приватная функция) для унификации формата SMILES (если нужно, но скорее нет), `normalize_name(name: str) -> str` (например, trim/upper for keys), `map_compound_ids(raw: dict) -> Dict` (функция, которая из сырого объекта ищет все возможные идентификаторы: InChIKey, CHEMBL, CID, и формирует структуру). Эти функции могут использоваться внутри transform стадий разных пайплайнов.
- `core/validation.py` – опционально, дополнительные проверки качества данных. Например, проверка, что у молекулы есть хотя бы один идентификатор, у таргета валидный UniProt (можно вызывать UniProt ID mapping API чтобы убедиться), или проверка соответствия типов взаимодействий заданному списку.

- **Именование классов и функций в core:** как обычно, PascalCase для классов (`PipelineRunner`), snake_case для функций (`run_all_pipelines`). Стараться использовать понятные имена.

- **Другие директории:**

- `src/bioetl/cli/`: здесь разместим один или несколько CLI-скриптов. Например, `run_pipeline.py` – скрипт, разбирающий аргументы (возможно с помощью argparse или Typer) и запускающий PipelineRunner. Можно сделать гибко: без аргументов – запускает полный ETL, или можно указать `--provider chembl --entity compound` для запуска отдельного сегмента, что удобно при отладке. CLI будет тонким слоем над core (т.е. он не содержит логики ETL, только вызов).
- `src/bioetl/utils/`: утилитарные функции, не привязанные к домену:
  - `rate_limit.py` – реализация ограничителя запросов (например, класс TokenBucketRateLimiter или простая pause-функция).
  - `retry.py` – декоратор или функция-помощник для повторного вызова функции при временной ошибке (HTTP 429 Too Many Requests, 500 Server Error).
  - `logging.py` – настройка логгера (формат логов, уровень). Хотя можно и не выносить, а настроить в core.
  - `parsing.py` – например, парсинг DOI из строки или PMID из DOI (маловероятно нужно, но вдруг).
  - В utils ничего специфичного по бизнес-логике химии, только общие вещи.
- `tests/`: набор тестов:
  - `tests/bioetl/clients/test_chembl_client.py` и аналогичные — модульные тесты клиентов (с мокированием HTTP-запросов, используя `responses` или встроенный httpx MockTransport).
  - `tests/bioetl/pipelines/chembl/compound/test_extract.py` — тест конкретного extract: можно подменить ChemblClient на заглушку, которая возвращает фиксированный JSON, и проверить, что extract функция собирает список нужной длины.
  - `tests/bioetl/pipelines/chembl/compound/test_transform.py` — подать на вход заранее заготовленный raw dict из ChEMBL и проверить, что на выходе `MoleculeSchema` заполнен корректно (например, name совпадает, InChIKey в верхнем регистре, не пустой и т.п.).
  - `tests/bioetl/pipelines/chembl/compound/test_load.py` — можно в тестовой БД (SQLite in-memory) проверить, что вставка проходит, уникальные ключи работают.
  - Интеграционные тесты: небольшой end-to-end на одном-двух примерах (например, взять ligandId из GtoP, пройти весь цикл и убедиться, что в итоге в БД есть запись).
  - Конечно, покрытие тестами – best practice, но учитывая однократный характер ETL, прижимисто возможно меньший приоритет, однако архитектура должна позволять тестировать компоненты независимо.

Резюмируя архитектуру: Проект будет состоять из четко разделенных модулей, отражающих этапы ETL. Такой дизайн (клиенты → схемы → пайплайны → БД) обеспечивает: - Возможность раздельно развивать и отлаживать интеграцию для каждого источника. - Переиспользование кода (общие функции, модели, клиенты). - Соблюдение принципа *single responsibility* для модулей (клиент только общается с API, transform-модуль только знает правила чистки/нормализации, storage знает только про БД). - Следование корпоративному стилю (структура `bioetl`), что упрощает чтение кода и автоматический контроль качества (CI сможет проверить соглашения именования, как было оговорено).

4. План по слоям: API-клиенты, нормализация, хранилище, пайплайны

Теперь распишем, как каждый слой будет реализован и взаимодействовать с другими, по шагам **ETL**:

A. API Clients Layer (слой клиентов API)

Задача этого слоя – предоставить удобные интерфейсы для внешних REST API, скрыв детали HTTP-запросов и форматов. Реализация: - **HTTP библиотека:** Используем либо `requests` (сессия на весь клиент для соединения keep-alive), либо `httpx`. `httpx` предпочтительнее, т.к. поддерживает как синхронный, так и асинхронный режим. Можно, например, использовать `httpx.AsyncClient` для тех сервисов, которые хотим вызывать параллельно, а `requests.Session` – для простых последовательных. Организуем в коде так, чтобы не смешивать подходы: либо все клиенты синхронны, а параллельность достигается потоками/процессами, либо клиенты имеют async-методы, а PipelineRunner использует `asyncio.run` для гонки coroutines. В MVP, чтобы упростить, можно начать с синхронного исполнения и без asyncio (избежать сложности дебага), но спроектировать код так, чтобы добавить async было несложно (например, httpx позволяет вызывать его API синхронно, используя `.run()`). - **Базовый класс `BaseClient`:** Можно внедрить общие атрибуты: `base_url`, возможно `rate_limit` (число запросов/сек), `api_key`. Этот класс может иметь встроенный `requests.Session` или `httpx.Client`. Он может предоставлять метод `_request(method, endpoint, **kwargs)` который выполняет вызов и возвращает `response.json()` (или выдает ошибку, если `status_code != 200`). Также тут можно реализовать: - Автоматическое ожидание при превышении лимита (например, замерять время между вызовами или использовать токен-бакет). - Обработку ошибок: если код 429 или 503 – делать паузу и повтор через несколько секунд (с экспоненциальной backoff). Количество повторов ограничим (например, 3 попытки, потом исключение). - Логирование URL и статуса (debug-лог). - **Конкретные клиенты:** Каждый сервис имеет свою специфику: - `ChemblClient`: знание о том, что нужно добавлять `.json` к URL, и о параметрах limit/offset. Например, метод `list_molecules(filters)`: будет дергать `GET /molecule?filter1=X&limit=...`. Также функция-генератор `iterate_all_molecules()` которая yield-ит порции по N штук: внутри делает цикл по offset до тех пор, пока не вернет пусто. - `PubchemClient`: методы `get_compound_by_inchikey(key)` -> вызывает `/compound/inchikey/{key}/cids/JSON`, получает список CID. Если список не 1, то надо как-то решать (InChIKey *должен* соответствовать единственному хим. веществу обычно). Затем `get_compound_details(cid)` -> `/compound/cid/{cid}/JSON?...`. Тут же можно сделать метод `bulk_get_compounds(cids list)` используя POST или через объединение ID в URL. - `IupharClient`: метод `get_all_ligands()` -> GET /ligands, `get_all_targets()` -> /targets, `get_interactions()` -> /interactions. Вероятно, раз JSON сразу список, можно не париться с пагинацией (если вдруг список не полный, надо проверить, но предположим полный). - `UniprotClient`: метод `get_protein(accession)` -> GET /uniprotkb/<acc>.json. Также можно сделать `get_proteins(accessions list)` – UniProt предлагает ID Mapping service, но он требует POST запроса с кучей ID и потом отдельно GET для получения результатов, что усложняет. В однократном сценарии может не окупиться сложность. Проще дергать последовательно (или параллельно 2-3 потока). - `PubmedClient`: метод `get_article_by_pmid(pmid)` -> EFetch call, возвращает JSON (надо парсить fields). - `CrossrefClient`: метод `get_work_by_doi(doi)` -> GET /works/doi. Внимание: DOI может содержать символы `/`, их надо URL-кодировать или использовать form: `/works/<DOI_PREFIX>/<DOI_SUFFIX>`. Проще, можно DOI вставить как `10.xx/yy` напрямую, но через HTTP-библиотеку надо закодить. Реализуем аккуратно. -

`OpenAlexClient`: метод `get_work(doi)` – у OpenAlex в URL можно прямо DOI подставить как `https://api.openalex.org/works/https://doi.org/<DOI>`. Нужно также encode, но можно использовать requests, он сам %-escape сделает если передать как параметр. - `SemanticScholarClient`: помимо GET, надо в заголовки положить `x-api-key`. Метод `get_paper(doi)` -> GET /graph/v1/paper/DOI:<doi>?fields=... (важно: encode DOI двояко, двоеточие и слеши тоже, но вероятно библиотека поможет). - **Возвращаемые данные:** Клиенты могут вернуть готовые Pydantic-модели (Pydantic позволяет валидировать из dict сразу). Например:

```
resp = self.session.get(url); data = resp.json()
return ChemblMoleculeRawSchema.parse_obj(data)  # где RawSchema это модель,
отражающая поля ответа
```

Однако, определять RawSchema для каждого — трудоемко. Можно возвращать просто `data: dict` и пусть transform-этап разбирается. Решим: **Клиент возвращает dict (JSON -> dict)**, а transform уже отдает нашим `Schema`. Исключение – GtoP interactions, где можно сразу вернуть list of dict (список взаимодействий). - **Особые случаи:** Если API возвращает *список с pagination*, клиент может скрывать пагинацию. Например, `ChemblClient.list_molecules()` может внутри делать цикл вызовов по 500 записей и склеивать большой список, но это может привести к огромной памяти. Лучше пусть выдает итератор (generator) или запишет сразу. Можно сделать: `for page in ChemblClient.iterate_molecules(page_size): yield page_data`. Тогда pipeline extract решит, сохранять все или писать по ходу. - **Тестирование клиентов:** Обеспечим, чтобы клиенты позволяли внедрить, например, кастомную http-сессию (для моков) или чтобы методы легко мокировались. Также важна корректная обработка ошибок: если сервис вернул 404 (не найдено) – метод может возвращать None или кидать свое исключение (например, `NotFoundError`). Эти исключения можно определить в `clients/errors.py` (например, `ApiClientError`, `RateLimitError`) и при необходимости отлавливать на уровне pipeline, чтобы решить, пропускать запись или падать.

B. Normalization & Transformation Layer (слой нормализации данных)

После получения сырых данных необходимо привести их к единой структуре и качеству – это происходит на этапе *transform/normalize* пайплайнов: - **Унификация полей:** Разные источники предоставляют разный набор атрибутов и в разных форматах. Наша задача – сконструировать объекты схем (`Schema`) заполненные консистентно. - *Пример (Molecule):* ChEMBL дает `molecule_chembl_id`, `pref_name`, `max_phase`, `molecule_structures` (sub-dict с SMILES, InChI, InChIKey), `molecule_properties` (logP, масса и т.д.), `cross_references` (список внешних ссылок, может содержать PubChem CID, Guide to Pharmacology ID и др.). PubChem дает `CID`, `IUPACName`, `Synonyms` (список), `MolecularFormula`, `IsomericSMILES`, `InChIKey` и т.д. GtoP ligand дает `ligandId`, `name`, `synonyms`, `chemblId`, `pubChemCID`. - Мы объединяем: `MoleculeSchema.chembl_id` = `molecule_chembl_id` из ChEMBL или из GtoP (если ChEMBL не вызывали, но GtoP дал id). - `pubchem_cid` = из PubChem или GtoP. - `iuphar_id` = ligandId из GtoP. - `name` = предпочитаем `pref_name` ChEMBL, если пусто – берем GtoP name, либо PubChem IUPACName. - `inchikey` = берем из InChIKey (ChEMBL или PubChem, желательно проверка что совпадают). - `smiles` = из ChEMBL или PubChem (желательно одно). - `max_phase` (если хотим) = из ChEMBL (говорит о стадии развития лекарства). - `synonyms` = объединяем списки синонимов из всех источников (например, GtoP предоставляет один основной name, PubChem много синонимов, ChEMBL может ID, trade names). - Здесь нормализация включает удаление дубликатов в списке синонимов, trim пробелов, возможно, выбор единого регистра для сравнения, но лучше

хранить как есть для информативности. - Также **deduplication**: если одна и та же молекула пришла из двух источников, на этапе transform можно обнаружить, что у них один InChIKey. Мы должны создать *один* объект `MoleculeSchema` для нее, объединив данные. Значит, нужно иметь механизм слияния: либо сразу в extract собирать уникальные, либо на этапе transform – например, использовать dict с ключом InChIKey или chembl_id. - Решение: на этапе сборки итогового списка молекул (после получения от ChEMBL и GtoP) мы пройдемся и сольем дубли. Возможно, имеет смысл отдельный шаг нормализации **после** transform разных источников — на уровне PipelineRunner, когда есть два списка молекул (от ChEMBL и от GtoP). Мы можем объединить их по ключу (InChIKey наиболее надежный для химии). - *Пример (Target):* - ChEMBL target API дает target_id (например CHEMBL612), название белка, может содержать список компонентов (список UniProt ID если это комплекс), тип таргета (single protein, protein family и т.д.). - GtoP target дает target id, name, gene symbol, species, и поле `uniprotId` (или несколько, если субъединицы). - UniProt дает полную информацию по одному accession: protein name, gene, organism, sequence (нам sequence не нужна для дашборда, но можно сохранить длину например). - Нормализация: - `TargetSchema.uniprot_id` – главный ключ (если есть несколько subunits, возможно, рассматриваем отдельные targets или берем главный? Лучше фокус на single proteins). - `name` – берем из UniProt (полное рекомендованное название) или GtoP (которое часто близко). - `gene` – из UniProt (основной ген), если нет – GtoP geneSymbol. - `organism` – унифицировать (латинское название или TaxID? UniProt дает и то, и то – можно взять Scientific Name). - `chembl_id` – если ChEMBL target существует и маппится на этот uniprot (ChEMBL target often corresponds to single protein by uniprot, in their component list). - `iuphar_id` – из GtoP if available. - `target_type` – класс (GPCR, kinase etc.) можно сохранить, GtoP явно классифицирует по типам, UniProt – по словесному описанию или GO классификация. В рамках dashboard полезно, но можно отложить. - Deduplication: уникальность таргета можно определять по **uniprot_id**. Если мишень не имеет uniprot (например, в ChEMBL могут быть комплекс или клетки, тогда uniprot нет) – можно фильтровать: вероятно, мы сконцентрируемся на белках, у которых есть унипрот. Если GtoP target – у него всегда должен быть uniprot (кроме случаев типа искусственных мишеней, но навряд ли). - Объединение: если один и тот же Uniprot встречается в ChEMBL и GtoP, мы объединяем. Используем dict с ключом uniprot. - *Пример (Interaction):* - ChEMBL activities: Каждая activity связана с chembl_molecule_id и с chembl_target_id (или target_chembl_id). Они также имеют поля типа измеренного эффекта (IC50, Ki, etc) и может быть ссылка на документ (поле `document_chembl_id` и можно запросить документ, но ChEMBL doc links often have DOI or PMID inside). - GtoP interactions: сразу ligandId + targetId + type + reference (PMID list). - Нормализация: - Представим внутреннюю модель `InteractionSchema` с полями: `molecule_id` (наш внутренний PK или chembl_id?), `target_id` (или target uniprot?), `type` (string, e.g. 'antagonist'), `source` (which DB gave it, e.g. 'GtoP' or 'ChEMBL'), `references` (list of pub identifiers). - Чтобы построить эти, нужно к моменту transform иметь mapping внешних IDs на внутренние: например, ChEMBL activity gives chembl compound ID and chembl target ID. Мы должны маппить: chembl compound -> наш MoleculeSchema (по chembl_id), chembl target -> наш TargetSchema (по uniprot if possible). Во время transform interaction можно обращаться к словарям, сформированным ранее, или пробрасывать ссылки. - Вероятно, pipeline для interactions будет запускаться **после** того, как transform молекул и таргетов выполнен, и мы имеем lookup: `chembl_id -> internal mol id` and `uniprot_id -> internal target id`. Но на этапе transform interactions, внутреннего ID еще нет, они появятся после load. Поэтому, возможны 2 подхода: 1. Привязывать interactions на этапе load, когда молекулы/таргеты уже в БД (можно выполнять SQL join на лету). 2. Или во время transform interactions, вместо внутренних PK, сохранить внешние (chembl_id, uniprot) и отложить разрешение до загрузки (заполнять внешними ключами через SQLAlchemy relationships). - Например, InteractionSchema можно сделать: `molecule_chembl_id` и `target_uniprot` (и/или target_chembl_id), плюс `reference_list`. А уже в load_interactions мы делаем поиск соответствующих Molecule.id по chembl_id (быстрый SELECT по индексу) и Target.id по uniprot, и тогда вставляем Interaction row с найденными FK. Если не найдены – логируем и пропускаем

(значит, вдруг чего-то нет, маловероятно). - *Пример (Publication):* - CrossRef/OpenAlex дают DOI, PubMed дает PMID, S.Scholar может дать либо то, либо по title. - Нормализация: - `doi` – основной ключ, если есть (у большинства статей есть DOI, кроме очень старых). - `pubmed_id` – если статья в PubMed, желательно заполнить (из NCBI или S.Scholar externalIds). - `title`, `journal`, `year`, `authors` – берем откуда получим. CrossRef и OpenAlex дают их сразу, можно использовать CrossRef как приоритет (он официально от издателя, значит наиболее точно), а OpenAlex/NCBI для проверки. - Если CrossRef недоступен для DOI, можно fallback на OpenAlex, если нет DOI – использовать PubMed by PMID. - Deduplicate by DOI или PMID. Если у статьи нет DOI (редко, но например до 2000х могли не иметь) – тогда PMID станет ключом. Но лучше сгенерить surrogate DOI (например, `pmid:12345`) чтобы ключ в таблице publication был уникальным в одном поле. - Extra: `citation_count` – можем заполнить из OpenAlex or S.Scholar для интереса, но не критично. - `url` – можно сформировать из DOI (https://doi.org/...), чтобы легко переходить.

- **Валидация и очистка:** Pydantic-модели помогут отловить грубые несоответствия типов. Например, если API вернул число, а мы ожидаем строку – Pydantic кинет ошибку, мы сможем логировать и, например, корректировать. Можно также задать регулярные выражения для некоторых полей (например, DOI формата `10.\d+/...` или InChIKey длины 27), Pydantic это позволяет. Все обнаруженные несоответствия следует либо поправить (если возможно), либо как минимум залогировать и пропустить запись.
- *Примеры нормализации:*
  - Приведение строк: trim пробельных символов, унификация регистра для ID (UniProt ID – верхний регистр, DOI – нижний регистр обычно, лучше привести к нижнему).
  - Удаление HTML-тегов или спецсимволов из текстов (например, названия статей могут содержать `&` – CrossRef может отдавать, надо декодировать).
  - Конверсия единиц: если бы загружали числовые данные (IC50 in nM vs μM), надо было бы приводить. У нас в основном структура, но если сохраняем значения активности, привести к единой единице (напр., nM).
  - Удаление дубликатов: мы уже рассмотрели – например, если молекула повторилась, убираем.
  - Слияние записей: объединение полей из разных источников (как обсуждали).
- **Инструменты:** В коде transform-этапов мы можем использовать вспомогательные функции, размещенные в core/utils. Например, `normalize_inchi_key(key: str) -> str` (привести к верхнему регистру, т.к. иногда м.б. строчные), `parse_authors(crossref_authors_list) -> List[str]` (из структуры CrossRef превратить в ["Surname, Name", ...]).
- **Выход transform:** На выходе каждого transform-модуля должны получиться **списки Pydantic-объектов** нашей унифицированной модели, готовых к сохранению. Например, `List[MoleculeSchema]`. Эти объекты можно сразу приводить к dict (через `.dict()`), но лучше передать в load как есть и использовать Pydantic или ORM интеграцию.

C. Storage Layer (слой хранения и работы с БД)

После нормализации данные нужно надежно сохранить в PostgreSQL, откуда ими смогут пользоваться аналитические инструменты. Особенности реализации слоя хранения: - **Схема БД:** Создадим несколько таблиц, отражающих ключевые сущности: - Таблица `molecule`: - `id` SERIAL PK - `chembl_id` VARCHAR, уникальный индекс (т.к. у известных соединений ChEMBL ID уникален, но может быть NULL если молекула только из PubChem). - `pubchem_cid` BIGINT, уникальный индекс (NULL если нет). - `iuphar_id` INTEGER, уникальный индекс (NULL если нет). - `inchikey` VARCHAR(27), уникальный индекс (InChIKey как универсальный идентификатор структуры, у каждого вещества должен быть, иначе строка "NA"). - `name` VARCHAR, `smiles` TEXT, `formula` VARCHAR(50) и прочие свойства по необходимости. - Можно хранить также

`drug_stage` (INT max_phase), `weight` (FLOAT), но это уже расширение по желанию. - Индексы нужны на основных идентификаторах для быстрых соединений: Inchikey индекс позволит быстро проверять дубликаты, chembl_id/pubchem_cid – для связи с внешними. - Таблица `target`: - `id` SERIAL PK - `uniprot_id` VARCHAR, уникальный (например, 'P12345' – max 10 символов). - `chembl_id` VARCHAR, индекс (многие белки имеют CHEMBL target ID, но не все, optional). - `iuphar_id` INTEGER, индекс (если есть в GtoP). - `gene` VARCHAR, `name` TEXT, `organism` VARCHAR(50). - `target_class` или `family` VARCHAR (GPCR, Ion Channel etc, optional). - Индекс основной по uniprot. - Таблица `publication`: - `id` SERIAL PK - `doi` VARCHAR, уникальный (можно ограничение UNIQUE, т.к. DOI уникален глобально). - `pubmed_id` INTEGER, уникальный (PubMed ID тоже уникален, но статья может иметь оба – один из них можно NULL). - `title` TEXT, `journal` VARCHAR, `year` INT, `authors` TEXT (либо отдельная таблица для авторов, но для дашборда можно просто строку). - `citation_count` INT (optional). - Индекс по pubmed_id, индекс по year (если часто фильтровать по году). - Таблица `interaction`: - `id` SERIAL PK - `molecule_id` INT FK -> molecule.id (с индексом) - `target_id` INT FK -> target.id (с индексом) - `interaction_type` VARCHAR(50) (тип действия: inhibitor, agonist, allosteric modulator... может быть NULL если неизвестно). - `sources` VARCHAR(20) (например, 'ChEMBL', 'GtoP' или 'Both'; или битовый флаг). Это чтобы отразить, откуда мы узнали про взаимодействие. - **Важно:** Один и тот же молекула-мишень могут встречаться в обоих ChEMBL и GtoP. Чтобы не дублировать в таблице, можно перед вставкой проверять: если такая пара уже есть, то обновить поле `sources` (дополнить). А если нет – создать новую. Иначе будет дублирование, мешающее аналитике. - Уникальный составной индекс (molecule_id, target_id) чтобы избежать дублей. - Можно добавить поле, например, `activity_value` и `activity_unit` для численного значения (например, IC50), но т.к. цель – скорее наличие связи, не обязательно детализировать. - Таблица `interaction_reference` (связь многие ко многим между interaction и publication): - `interaction_id` INT FK -> interaction.id - `publication_id` INT FK -> publication.id - PK составной (interaction_id, publication_id). - Здесь храним, какие публикации подтверждают данное взаимодействие. GtoP часто даёт одну главную ссылку, ChEMBL может иметь много (или вообще откуда-то взятая инфа). Если не указывать, то получится, что мы знаем о взаимодействии, но не указываем источник. Для knowledge graph важно указать, чем подтверждено. - Заполнять эту таблицу будем, имея списки PMID/DOI из источников. Если ни один источник не дал реф для взаимодействия, можно оставить без рефа (но такое редкое). - **SQLAlchemy ORM:** - Определим классы Python, соответствующие таблицам (как описано выше). Для связей можно настроить relationship: - `Molecule.targets` (backref many-to-many through interactions) – но можно и не делать сложных relationships, а оперировать вручную через join. However, relationships полезны если будем навигировать объектно. - `Interaction.publications` as relationship secondary=interaction_reference. - ORM моделей имена: `Molecule`, `Target`, `Publication`, `Interaction`, `InteractionReference`. PascalCase, как требует стиль, и `__tablename__` lowercase plural or singular (to choose consistent, e.g. molecule or molecules). - **Связывание с Pydantic:** - Pydantic может создавать модели на основе ORM (`Config.orm_mode = True` in v1 or `model_config = {'from_attributes': True}` in v2). Мы можем использовать это, но в нашем случае проще: Pydantic -> dict -> ORM. - Процесс `load_*` функций: - Например, `load_compounds(list[MoleculeSchema])`:

```
objs = []
for schema in list_schemas:
    data = schema.model_dump()  # (in Pydantic v1: schema.dict())
    # optionally remove None values for not null columns or pop relational
fields
    obj = Molecule(**data)
    objs.append(obj)
```

```
session.bulk_save_objects(objs)
session.commit()
```

Bulk operations быстры, но нужно быть осторожным с size (можно батчить по 1000). - Если нужно получить id вставленных (например, для взаимодействий), можно после коммита сделать `session.refresh(obj)` или query. Но лучше после вставки молекул и таргетов, сделать маппинг вручную: - Например, составить dict `chembl_id -> molecule.id` и `uniprot -> target.id` после их вставки. Можно за один SELECT вытянуть. Это понадобится для связывания interactions. - **Обработка транзакций:** Каждая группа вставок (по сущности) может быть отдельной транзакцией. Например, сначала вставили все molecules (commit), затем targets (commit). Это упростит откат в случае сбоя на определенном этапе (не затронет предыдущие). Но если один из этапов падает, БД окажется частично заполненной. Для однократной загрузки это приемлемо (можно почистить и перезапустить). Альтернатива – обернуть весь процесс в одну транзакцию и коммит в конце, но это невозможно, т.к. объем данных большой, да и связи требуют видеть уже сгенерированные id. - **Инициализация БД:** В проекте можно иметь `alembic` миграции или просто скрипт создания. Для MVP можно вызывать `Base.metadata.create_all(engine)` при первом запуске (если БД пустая). Позже стоит накатить миграции (например, если добавятся поля). - **Performance:** С точки зрения БД, объем данных: молекул (несколько тысяч, если берем ограниченно), таргетов (сотни), интеракций (несколько тысяч), публикаций (тоже тысячи). Это небольшие размеры для PostgreSQL, все операции должны быть быстрыми. Стоит добавить индексы на внешние ключи (Postgres сам индексирует PK, но FK – нет, если нужны фильтры). Например, часто будем искать взаимодействия по target_id или molecule_id, так что индексы на них важны. - **Dashboards usage:** В итоге, таблицы спроектированы для типичных запросов аналитики: - Список всех молекул с их свойствами. - По конкретной молекуле – список мишеней, на которые она влияет, с указанием типа влияния и источника, а также ссылок на публикации. - По таргету – список соединений, которые на него действуют. - По публикации – какие взаимодействия она описывает. - Эти запросы осуществляются простыми JOIN-ами между 3-4 таблицами. - Дополнительно можно построить представления (view) для удобства, но это уже по желанию аналитиков. - **Визуализация связей:** Если потребуется визуализировать граф, возможно, проще выгрузить эти таблицы и использовать графовые инструменты, но хранение в Postgres вполне достаточно для числовых дашбордов (например, сколько мишеней у препарата, и т.д.).

D. Pipeline Orchestration Layer (слой пайплайнов и оркестрации)

Этот слой связывает всё вместе, определяя *последовательность и логику выполнения* ETL-этапов: - **Оркестратор PipelineRunner:** - Этот компонент (в `core` или `pipelines/run_pipeline.py`) будет точкой входа для запуска всего процесса. Он может быть реализован как класс `PipelineRunner` с методами `run_all()` или как набор функций. Выберем класс для удобства, с возможностью настраивать какие пайплайны включены. - Ему нужны ссылки на необходимые клиенты и на соединение с БД. Решение: - Инициализируем нужные клиенты (ChemblClient, etc.) с необходимыми параметрами (например, `ChemblClient(base_url, rate_limit=1)`, `SemanticScholarClient(api_key=XYZ)`). - Открываем сессию БД. - Выполняем этапы: 1. **Extract & Transform Molecules:** - Вызвать `iuphar/ligand/extract` → получим list ligands (из GtoP). - Вызвать `chembl/compound/extract` (возможно с фильтром: только CHEMBL IDs из списка GtoP или max_phase>=3 и т.д.) → получим list ChEMBL compounds. - Вызвать `pubchem/compound/extract` – либо сразу внутри хода: по списку InChIKey из предыдущих, получить доп. свойства. Или организовать так: сначала объединим ключи, потом дернем PubChem. - Объединить результаты: возможно, сделать функцию `merge_molecules(gotp_list, chembl_list, pubchem_list)` – возвращающую уникальный list, с объединенными полями. (Логика

объединения как обсудили: по InChIKey мержить). - Прогнать через `transform.molecule(list_raw)` – получим List[MoleculeSchema]. - Вызвать `load.molecule(list_schema)` – запишет в БД. После коммита, можно сохранить mapping chembl_id->mol.id и inchikey->mol.id, для использования далее. 2. **Extract & Transform Targets:** - `iuphar/target/extract` → list targets (GtoP). - `chembl/target/extract` (ChEMBL Target dictionary, хотя если фокус только на GtoP/ChEMBL intersections, можно выгружать только нужные). - Тут оптимизация: ChEMBL target – их тоже много, лучше не тянуть все. У нас есть список целевых Uniprot из GtoP, а ChEMBL target можно получить только если надо дополнительную инфу (в принципе UniProt сам даст достаточно). - Поэтому, возможно, ChEMBL target extract вообще не нужен, если у нас есть UniProt + GtoP. Разве что, чтобы получить CHEMBL target ID, но он нам не нужен особо, кроме как идентификатор связи. Но можно сохранить его в TargetSchema. - `uniprot/protein/extract` – вместо вытягивания всех, можно вытянуть конкретные accession, которые у нас есть (из GtoP list). Т.е. прочитать unique set of uniprot from GtoP targets, и вызывать UniprotClient.get_protein для каждого (параллельно или по одному). - Объединить: слияние по uniprot (UniProt as main reference). - `transform.target(list_raw)` – List[TargetSchema]. - `load.target(list_target_schema)` – insert. Сохранить mapping uniprot->target.id. 3. **Extract & Transform Interactions:** - `iuphar/interaction/extract` → GtoP interactions list (each with ligandId, targetId, ref(s)). - `chembl/activity/extract` → можно сделать: пройти по списку mol или target, или использовать ChemblClient to get all activities for needed chembl_ids. - Здесь если мы ограничились molecules to those in GtoP or some set, то и активности надо только для них. Хорошо: ChEMBL activity API позволяет фильтровать, например `?molecule_chembl_id__in=C1,C2,...` (нужно проверить, но обычно есть фильтр __in). Если нет, то цикл по molecules: для каждого молекулы дернуть ее activities (ChEMBL might have thousands per mol, but GtoP ligands обычно не настолько исследованы). - Это может быть долго, возможно, ограничиться значимыми: например, activities с standard_type "IC50" или Ki etc. Но лучше взять все. - Объединить interactions: - Перевести внешние ID в наши ID: GtoP: ligandId->chembl_id (нужно получить chembl_id соответствующий ligandId: GtoP ligand extract уже дала mapping ligandId->chembl_id, или ligand name. В GtoP ligand JSON, как сказано, есть chemblId). - Если GtoP ligand имеет chemblId, то можем конвертировать ligandId -> chembl_id -> найти Molecule через mapping chembl->id. Если нет chemblId (бывает ли? может для некоторых endogenous?), тогда пробовать InChIKey match с молекулой (GtoP ligand info может иметь InChIKey? Если не ошибаюсь, GtoP ligand JSON содержит smiles, можно инчикей вычислить, но скорее chemblId должен быть у большинства). - GtoP target: targetId -> uniprot (GtoP target JSON likely has uniprot or we correlate by name/gene which is riskier). Скорее, GtoP target JSON does include a `uniprotId` field for each target. Если да, то mapping targetId->uniprot -> find Target id. - ChEMBL activity: provides `molecule_chembl_id` and `target_chembl_id` (or target protein accession in some field?). Possibly they provide target as multiple target components (like for a protein complex). - Simplify: if activity target type is single protein, it often includes `target_components` list with accessions. If one accession, use it. If multiple, skip or treat separate. - Or use target_chembl_id: need to map to uniprot: We could have from ChEMBL target extract a mapping target_chembl_id -> uniprot. Possibly we skip by using only GtoP targets? Might miss some if GtoP doesn't list all. - Alternatively, for each ChEMBL activity, we could call `GET /target/<target_chembl_id>` to get the protein components, but that's heavy if many. Alternatively, we might have pre-fetched all ChEMBL targets for our uniprot set (if any left). - Or cross-check: if uniprot in our target list, fine; if not, maybe ignore that interaction as out-of-scope (since we limited molecules and targets). - После получения взаимодействий из обоих, объединяем: - Если одна и та же пара (mol,target) встречается и там, и там, объединяем источники. - Все ссылки на ref собираем в единый список. - Результат transform: список InteractionSchema (пока с внешними ссылками resolved: i.e. now containing molecule_id and target_id as internal integers, plus references list of (doi or pmid)). - `load.interaction(list_interactions)` – пишем в таблицу `interaction` и `interaction_reference`: - Реализация: проходим по каждому InteractionSchema: - Получаем (or

already have) int mol_id, int tgt_id, type, sources. - Вставляем или обновляем `interaction`: - Если уникальный ключ (mol_id,tgt_id) конфликт (ON CONFLICT in postgres) – тогда обновить sources = объединение старого и нового (и, возможно, union of reference lists). - Лучше, однако, на уровне Python убрать дубли: например, использовать dict {(mol_id, tgt_id): combined_info}. - Commit interactions. - Затем связки references: для каждой InteractionSchema.references (список pub_id или doi) найти publication_id: - Если публикация уже загружена? На данный момент мы еще не грузили publications. Поэтому, либо сначала загрузить publication, либо сначала interaction references, потом resolve pubs. - Может быть логичнее: собрать все уникальные references здесь в список to-fetch, но саму связь пока не заполнять. - Решим так: на шаге interactions собираем *полный список уникальных публикаций (doi/pmid)* которые нам понадобятся. Сохраняем, а заполнение `interaction_reference` сделаем после загрузки всех publications. - Итак, interactions грузим без references, но сохраняем mapping interaction (mol_id,tgt_id) -> [list of references]. 4. **Extract & Load Publications:** - Из предыдущего шага у нас список уникальных PubMed ID/DOI (и, потенциально, GtoP references, которые скорее всего тоже PMID, иногда DOI). - Для каждого идентификатора: - Если есть DOI: использовать CrossRef/OpenAlex. Можно делать это параллельно, т.к. overhead небольшой. - Если только PMID (без DOI): воспользоваться NCBI EFetch или S.Scholar to get DOI. PubMed EFetch дает title, etc, но DOI тоже часто в поле article IDs есть. Если есть email, EFetch можно до 10/sec. - Strategy: - Try CrossRef by DOI (likely covers most). - If no DOI but have PMID, use NCBI to get at least title/journal/year. Possibly also it might give DOI in XML. Actually, yes, PubMed eutils often includes `<ArticleId IdType="doi">...</ArticleId>`. - If we want to be thorough: use S.Scholar only if DOI not found via NCBI. - Build list of PublicationSchema. - `load.publication(list_pub)` – insert all into publication table. Use unique constraints to avoid duplicates (if two interactions referenced same paper, we only insert once). - After insertion, get mapping for reference resolution: e.g. a dict from (doi or pmid) to publication.id. 5. **Load Interaction References:** - Now loop through stored interaction->references mapping: - For each (mol_id, tgt_id) pair, find its interaction.id (by querying the table, or maintain a dict from earlier insertion if we captured the new ID via returning). - If using Postgres with `ON CONFLICT DO NOTHING` or so, we might not easily get the ID. Simpler: fetch back via SELECT by (mol_id, tgt_id) which is indexed. - Then for each reference in list: - find publication_id via dict (if not found, skip). - insert row into interaction_reference (or bulk insert for all). - Commit. 6. **Cleanup:** Закрыть сессию, вывести финальные логи (сколько записей в каждой таблице, время выполнения). - Вся эта последовательность может быть разбита на подзадачи/подпайплайны, но концептуально так. Для удобства, PipelineRunner может иметь методы `load_molecules()`, `load_targets()` etc., вызываемые по порядку.

- **Конфигурация pipelines:**
- Как было отмечено, можем иметь конфиг-файлы на pipeline. Например, `configs/pipelines/chembl_compound.yaml`:

```
provider: chembl
entity: compound
filters:
  max_phase: 3
  has_assay_data: true
```

– который использует Chembl API query parameters, чтобы не тащить все молекулы.

  - PipelineRunner мог бы читать такие конфиги и передавать параметры в extract функции (например, `extract_all_chembl_compounds(filters=...)`).
  - Для MVP не обязательно усложнять, но предусмотрим возможность.

- **Параллелизация:**

- Слои архитектуры позволяют некоторые вещи делать параллельно для ускорения. Например:
  - Выгрузка молекул и таргетов (Chembl vs GtoP vs PubChem) можно параллельно, т.к. независимы.
  - Запросы UniProt для разных accession – можно запускать concurrency (multi-thread or asyncio gather).
  - Запросы CrossRef/OpenAlex для разных DOI – параллельно.
  - Но, например, запись в БД желательно делать последовательно для целостности (или в отдельных транзакциях).
- Можно воспользоваться concurrency библиотеками: `concurrent.futures.ThreadPoolExecutor` для I/O-bound tasks (like many HTTP calls). Или использовать asyncio with httpx which internally uses connection pooling and concurrency.
- В архитектуре, мы можем скрыть параллельность за интерфейсами:
  - Например, `PubchemClient.bulk_fetch(cids)` внутри себя запускает параллельно 5 потоков, но наружу это не видно.
  - Или PipelineRunner может применять `asyncio` to gather calls: e.g., gather multiple PubMed fetches.
- Главное, чтобы параллелизм не нарушал лимитов API: будем ограничивать согласно обсуд. (Напр., semaphore of 3 for NCBI calls).
- Это скорее optimization step, но хорошо иметь заложенную структуру (e.g., code design that can easily switch to async without rewriting everything).

- Если **MVP** — можно всё последовательно сделать (будет медленнее, но проще), а оптимизацию concurrency отнести на этап улучшения.

- **Error handling и устойчивость:**

- Если какой-то один источник временно недоступен или возвращает ошибку, решения:
  - В MVP: пусть падает (чтобы мы увидели проблему).
  - Позже: Можно имплементировать повторные попытки (Retry) и таймауты.
  - Например, если PubChem не ответил, можно сделать 3 ретрая, потом пропустить те CID (потеряем часть данных, но основной процесс продолжится).
  - Если UniProt не отвечает для какого-то белка — залогировать и пропустить.
  - Если целый сервис down — пайплайн можно запустить без него (должна быть опция отключить источник).
- **Atomicity:**
  - Т.к. это one-off pipeline, нет большой нужды в сверхсложной отказоустойчивости. Но если в середине процесса упало, база частично заполнена. Возможно, лучше при каждом запуске чистить таблицы и загружать с нуля, чтобы получить консистентный срез.
  - Это можно автоматизировать: PipelineRunner перед началом может очищать target tables (или лучше через `TRUNCATE CASCADE` в правильном порядке).
  - Но если БД уже используется для дашборда, возможно, нежелательно оставлять ее пустой надолго. Тогда можно загружать во *временные* таблицы и потом переключать схему (это уже сложнее, скорее лишнее).
- **Logging/Monitoring:**
  - В проект встроим журналирование. В консоли или файле должны отображаться ключевые события:

- "Fetched 250 compounds from ChEMBL (page 1)",
- "Transformed 500 molecules, merged into 480 unique after deduplication",
- "Inserted 480 molecules into DB",
- "Fetched 300 interactions from GtoP, 1200 from ChEMBL",
- "Total unique interactions 1300; with references 1100 have at least 1 reference",
- "Fetched metadata for 100 publications (DOI), 50 publications (PMID via NCBI)".
- И т.д. Такие логи помогают отследить прогресс и проверить корректность этапов.

В итоге, слой пайплайнов координирует работу всех предыдущих слоев, обеспечивая, чтобы данные прошли путь от API до БД в нужном порядке и виде. Благодаря тому, что проект структурирован по источникам/сущностям/этапам, можно запускать как все сразу, так и выборочно, не нарушая целостность: например, можно обновить позже только публикации или только перезапустить ChEMBL-пайплайн, если появилась новая версия ChEMBL.

5. План внедрения по этапам (от MVP до финального состояния)

Реализацию проекта целесообразно разбить на несколько этапов, постепенно наращивая функциональность и сложность. Это позволит получить работающий прототип (MVP) в кратчайшие сроки, а затем итеративно улучшать его, контролируя риски.

Этап 0: Проектирование (текущий этап). Создание архитектурного плана, согласование структуры и требований (то, что мы делаем сейчас).

Этап 1: MVP – Базовая интеграция основных сущностей. - **Цели MVP:** собрать минимально полезный сквозной сценарий: загрузить небольшое подмножество данных (например, несколько молекул и мишеней) и сохранить в БД, убедиться, что все слои (клиент → схемы → БД) работают вместе. - **Реализация:** - Начать с **Guide to Pharmacology** как источника небольшого объема но релевантных данных. Реализовать `IupharClient` и выгрузить, скажем, 10–20 лигандов и таргетов (можно фильтровать по чему-нибудь или взять первые N). - Реализовать `UniprotClient.get_protein` для получения деталей 1-2 мишеней из списка. - Реализовать базовые схемы `MoleculeSchema`, `TargetSchema`, `InteractionSchema`. - Реализовать простейший PipelineRunner, жестко вызывающий: get 10 ligands -> transform -> load; get targets -> transform -> load; (взаимодействия GtoP можно сразу из ligands/targets получить или отложить). - Настроить соединение с локальной PostgreSQL (или SQLite для тестов) и убедиться, что таблицы создаются и заполняются (проверить, что можно выполнить SQL-запросы и увидеть данные). - **Проверка:** На этом шаге должна получиться таблица molecules с записями, таблица targets, возможно interactions (можно одну-две связи вручную добавить или через GtoP). - **Упор на структуру:** Здесь важна правильная организация каталогов, имен, чтобы следовать `bioetl` style. Настраивается базовый CI (например, запуск линтера, проверка импортов). Можно подключить `pre-commit` с flake8, black, а главное – скрипт проверки naming (из `11-naming-policy.md`) чтобы с самого начала соответствовать требованиям.

Этап 2: Расширение охвата источников и связей. - **Цель:** добавить остальные источники один за другим, интегрируя их с уже полученными данными. - **Подзадача 2.1: ChEMBL integration (compounds & activities).** - Реализовать `ChemblClient` с базовыми методами (lookup, get molecule by ID, list molecules by filter). - Настроить в конфиге или коде фильтр для ChEMBL: например, взять все молекулы, у которых `max_phase >= 4` (то есть одобренные препараты) – это порядка 2000 молекул, приемлемо. Или, как вариант, взять все moleculs из GtoP (из Этапа 1) – их chembl_id мы знаем, можно дернуть конкретно их. - Реализовать пайплайн `chembl/compound`

extract/transform/load: загрузить эти молекулы. Слить с уже имеющимися (GtoP) — здесь проверяется механизм deduplication и объединения данных. - Реализовать пайплайн `chembl/activity`: выгрузить взаимодействия для тех же молекул (ChemblClient может по каждому chembl_id вызвать `/activity?molecule_chembl_id=...`). Т.к. число молекул ограничено, активностей будет не запредельно много. Преобразовать их в InteractionSchema (маппинг chembl_id->internal id, target_chembl_id -> target via uniprot). - Проверить: записи в таблице interaction появились, ссылки на publication пока будут в виде идентификаторов (например, ChEMBL `document_chembl_id` – его нужно будет сконвертировать в DOI, но пока можно проигнорировать или отметить). - **Подзадача 2.2: UniProt enrichment.** - Если еще не, реализовать `UniprotClient` + `uniprot/protein` pipeline: пройтись по всем target uniprot_ids, которых мы набрали (из GtoP и ChEMBL activities), и получить их подробности. Обновить таблицу `target` полями organism, full name. - Если при Этапе 1 мы занесли targets только из GtoP (которые уже имели имя/ген), то теперь дополним/обновим их из UniProt. Это можно сделать либо через `UPDATE` запросы, либо при загрузке можно применить `ON CONFLICT DO UPDATE`. Например, мы можем делать upsert по uniprot_id. - Убедиться, что все targets имеют заполненные core поля (organism, gene, etc). - **Подзадача 2.3: PubChem enrichment.** - Реализовать `PubchemClient` + `pubchem/compound` pipeline: для всех молекул, у которых есть InChIKey (а это должны быть все), получить PubChem CID и основные свойства (molecular formula, maybe weight, synonyms). - Обновить таблицу `molecule`: установить поле formula, может weight. Т.к. запись уже вставлена, здесь снова либо upsert, либо update. Можно написать отдельный метод `update_molecules_with_pubchem(session, data)` который пройдется по списку и обновит соответствующие строки. - (В архитектуре, можно было сначала все собрать а потом вставить – но мы выбрали поэтапную вставку, так что обновления — нормальная вещь. Либо можно было отложить insert molecules до получения всех данных. Но ничего, сделаем update.) - Проверить: молекулы теперь обогащены, дубликаты не размножились. - **Подзадача 2.4: References integration (Publications).** - До этого момента, взаимодействия ChEMBL и GtoP имеют ссылки (ChEMBL document IDs, GtoP PMID). Реализуем сбор публикаций: - Собрать все уникальные PMID из GtoP (они сразу были, GtoP references likely list PMID). - Собрать DOI из ChEMBL документов: придется запросить ChEMBL документ API: e.g. `/document/{doc_id}`. Возможно, лучше сначала посмотреть: ChEMBL activity возможно прямо содержит поле `doi` или `published_year`. Если нет, то надо вызвать /document. Чтобы не грузить все, можно ограничиться: у нас ограниченное число activities, у них doc_ids, дернем по каждому. ChEMBL doc тоже API 1/sec, но количество doc <= count(activities) which is manageable. - Реализовать `crossref/publication` и/или `pubmed/publication`: - Для каждого рефа: если есть DOI (from ChEMBL or if GtoP gave DOI maybe?), вызвать CrossRef API. - Если только PMID, вызвать NCBI. - Сконструировать PublicationSchema. - `load.publication` – вставить все уникальные publications. - `load.interaction_reference` – теперь создать связи. - Проверка: таблица publications заполнена (с правильными title etc., можно ручным выборочным способом сравнить с PubMed). - После этого, вся цепочка от молекулы->мишени->ссылки представлена в базе. Это уже полноценный граф знаний. - **Подзадача 2.5: Semantic Scholar (опционально).** - При необходимости, интегрировать S.Scholar: например, получить citation_count для каждой DOI. Это не критично, но можно добавить: - `semanticscholar/publication/extract`: по списку DOI (или S2PaperID, но лучше DOI) — но их может быть много. Надо учесть rate limit 1/sec. Можно сильно ограничить, например, только для 100 самых цитируемых? Но ладно. - Возможно, отложить S.Scholar к этапу 3 (улучшения). - Завершение Этапа 2: Теперь проект умеет соединять все данные вместе для заданного поднабора сущностей. На этом этапе стоит провести **тестирование**: - Юнит-тесты для ключевых функций (особенно transform). - Интеграционный тест: например, взять одну известную связку из литературы (например, Аспирин – Циклооксигеназа, известное взаимодействие, с PMID), и проверить, что после прогонки пайплайна эта связка есть в базе и содержит ту самую публикацию. - Проверить, что при повторном запуске (на чистой базе) всё отрабатывает без

ошибок, и что при изменении параметров (например, лимита на молекулы) pipeline корректно реагирует.

Этап 3: Финализация и расширение. - **Цель:** довести объем данных и качество интеграции до требуемого финального состояния, а также обеспечить надежность и задокументированность. - **Задачи:** - Расширить объем: если на этапе 2 мы ограничивали по фазе или списку, решить, нужен ли полный охват. Если да – можно попробовать увеличить границы (например, max_phase >= 1 для ChEMBL, это десятки тысяч соединений – возможно, приемлемо; или все GtoP + их близкие аналоги). - Оптимизировать производительность: - Внедрить asyncio или многопоточность для сетевых вызовов, где это безопасно. Например, реализовать `ChemblClient.get_many(ids_list)` с asyncio.gather (по 1 req/sec per task, но можно интерлейсить разные tasks – не сильно ускорит Chembl, но PubMed и CrossRef можно дергать 5-10 параллельно). - Добавить **асинхронный** PipelineRunner: т.е. `async def run_all()` и в нем `await` разных calls. Или на уровне отдельных extract: e.g. `extract_pubmed_publications` can fetch all concurrently. - Проверить, что при больших объемах памяти хватает и скорость удовлетворяет. Если нет – добавить постраничную обработку (например, молекулы ChEMBL: загружать по 500, сразу писать в БД, освобождать память). - Улучшить отказоустойчивость: - Встроить декоратор `@retry` для внешних вызовов. - Обработку исключений: напр., если один DOI не нашелся в CrossRef (404) – логировать предупреждение и продолжать, создавая запись с минимальной инфой (можно title как "Unknown" или пропустить reference). - Оградиться от некорректных данных: например, если UniProt не содержит имени (вдруг), то взять из GtoP. - **Документация:** Подготовить документацию (раздел `docs/`), описывающую: - Как запустить pipeline (с примерами командной строки). - Структуру базы (схема таблиц, связи). - Описание, какие источники и какие поля интегрированы. - Возможные настройки (например, файлы конфигурации pipeline). - Это полезно для команды аналитиков, кто будет пользоваться витриной. - **Дополнение тестов:** Покрыть оставшиеся модули. Особенно проверить corner cases (например, молекула без InChIKey, взаимодействие без PMID). - **Код-ревью и соответствие styleguide:** Проверить, что все имена файлов, классов, функций соответствуют `11-naming-policy.md` (можно запустить внутренний инструмент `bioetl-naming-lint` при его наличии). Поправить отступы, типы (можно добавить `муру` статическую проверку). - **Finishing touches:** Убедиться, что нет чувствительных данных (ключей) в коде – лучше их ожидать через переменные окружения или конфиг вне гита. Настроить `.env` или config for API keys (Semantic Scholar). - **Результат:** Полностью функционирующий ETL-процесс, который по запросу (запуск CLI) за один прогон заполняет PostgreSQL свежими агрегированными данными. Это и будет финальное состояние проекта.

Этап 4: (Опционально, после финала) Пилотное использование и отзыв. - Запустить ETL на реальных данных (например, полный список GtoP + phase>=3 ChEMBL) – оценить время выполнения (может быть несколько часов из-за API ограничений, надо быть готовым). - Дать доступ аналитикам к БД, собрать обратную связь: хватает ли данных, корректно ли связалось. - При необходимости, выполнить доработки (например, добавить поля, о которых не подумали, или дополнительный фильтр). - Зафиксировать версию 1.0.

6. Возможные риски и расширения проекта в будущем

При реализации такого интеграционного проекта следует учитывать потенциальные **риски** и закладывать возможности **расширения** на будущее:

- **Риск 1: Ограничения и нестабильность внешних API.** Все наши данные зависят от сторонних сервисов. Возможны ситуации:

- API сервис временно недоступен (Downtime) или отвечает с ошибками. Если это случится во время нашего единовременного забора, процесс может не завершиться корректно. *Митигировать:* реализовать устойчивость – повторные попытки, тайм-ауты, а также логирование, какие ID не удалось получить (чтобы потом отдельно догрузить при надобности).
- Изменение API: например, к 2026 году формат JSON ChEMBL может поменяться или URL. *Митигировать:* изолировать такие детали в клиентах и конфигурации, чтобы правка заняла минимум времени. Регулярно обновлять документацию по API.
- Лимиты запросов: мы уже заложили throttle, но все же, если объем данных будет расширен, есть риск упереться в лимиты (особенно ChEMBL 1req/sec). *В будущем*, если нужен частый и масштабный доступ, можно **рассмотреть переход на локальные копии** (например, развернуть базу ChEMBL локально или использовать ее FTP-дамп). Пока условие запретило дампы, но это вариант расширения, если проект вырастет и потребуется обновлять данные чаще.
- Semantic Scholar: если вдруг потребуется массово получить данные, базового лимита может не хватить. Можно подать заявку на повышенный лимит или использование их платного API. Альтернатива – привлечение других источников (например, MAG/OpenAlex достаточно).

- **Политика лицензирования и коммерческого использования:** GtoP упоминала про коммерческое использование (финподдержка). Если наш проект коммерческий, нужно следить за лицензиями. Данные в основном ODbL (ChEMBL, GtoP) и CC BY (UniProt), OpenAlex (CC0), CrossRef (подразумевается free). Надо просто соблюдать атрибуцию в документации.

- **Риск 2: Большие объемы данных и время выполнения.**

- Если вдруг решено тянуть **все** данные (например, весь ChEMBL с >2 млн соединений, >15 млн биоактивностей), то REST API безумно долго будет работать (месяцы). В нашем сценарии, скорее всего, бизнес-логика ограничит рамки (например, рассматриваем только «drug-like» молекулы или только интересующие классы).
- Но если появится требование обновлять регулярно и полно, лучше перейти на гибрид: крупные базы (ChEMBL, PubChem) держать локально (скачивать дампы), а маленькие (GtoP) тянуть API. То есть в будущем архитектура может **расшириться**: добавить новый тип источника – **базы данных** (например, подключение к локальной копии через SQL или **GraphQL** API если сервис предоставляет).
- В текущей архитектуре, подключение нового источника – просто создание нового клиента и пайплайна. Например, можно интегрировать **DrugBank** (если есть ключ) или **PDB** (Protein Data Bank) для 3D-структур. Это потребует расширить схему (таблица для структур и связей с мишенями), но принципы те же.

- **Оптимизация времени:** Если запуск даже ограниченного набора занимает слишком много (скажем > 5 часов), можно обдумать распараллеливание на уровне процессов или распределенно. Например, разбить список молекул на части и запускать несколько инстансов скрипта параллельно (с разными offset, но аккуратно с API-limit). Инфраструктурно – поднять временно несколько VM, каждая тянет часть данных, потом объединить в БД. Это вне рамок разработки (скорее DevOps план), но возможно.

- **Риск 3: Интеграция и качество данных (Data Quality).**

- Возможны несоответствия: одна база может называть вещество X, а другая Y, и мы можем не всегда понять, что это одно и то же (если, скажем, нет InChIKey, или разные формы соли/изомеры).
- Мы выбрали InChIKey и UniProt как «истинные идентификаторы». Но:
    - Разные InChIKey для разной соли одного лекарства — мы тогда будем считать их разными молекулами, хотя по смыслу, возможно, хотели объединить. Это тонкий момент: e.g. ChEMBL separate CHEMBL IDs for salt forms. GtoP, возможно, оперирует базовой формой.
    - В дальнейшем можно добавить этап **нормализации химической идентичности**: например, использовать сервис UniChem (EBI) для маппинга идентификаторов, или сравнивать InChI (не key, a full InChI to unify tautomers).
    - Пока это вне объема, но можно задокументировать: «в текущей реализации вещества различаются по InChIKey (учитывает четко определенную структуру). Если нужно агрегировать на уровне активной формы, необходим дополнительный слой нормализации химических структур.»
- Возможны дубликаты записей: мы постарались их слиянием убрать. Но если всё же что-то прошло (например, один таргет без Uniprot, другой с, оказались тем же геном) – потенциально, у нас будет две записи target. Такое может быть, если ChEMBL target – комплекс без uniprot, а GtoP имел отдельные subunits. Мы можем либо игнорировать комплекс, либо добавить у target флаг типа.

- **Митигировать качество:**

    - Добавить **пост-валидацию**: пройтись по загруженным данным и найти потенциальные проблемы. Например, скрипт QC: проверить, что для каждого interaction есть хотя бы одна publication reference (если нет – значит взаимодействие плохо подтверждено, можно пометить).
    - Проверить, что не появилось молекул без названия, targets без gene. Если есть – попытаться заполнить (например, если target – комплекс, можно хотя бы имена субюнитов перечислить).
    - Такие инструменты можно хранить в `qc/` или `tools/` согласно структуре (например, `tools/data_quality_report.py` который генерит отчет).

- **Будущие расширения:**

- **Инкрементальные обновления:** Сейчас пайплайн разово берет snapshot. В будущем, если потребуется актуализировать данные (скажем, раз в квартал):
    - Можно модифицировать клиенты API для поддержки фильтра "updated_since". Некоторые API (CrossRef, OpenAlex) позволяют запрос только новых записей по дате. ChEMBL API, возможно, нет явного updated_since, но выпускаются новые версии периодически. GtoP обновляется периодически (версии), можно снова все тянуть, не так много.
    - Проще всего – перезагружать все заново, учитывая умеренный объем. Но если объем станет большой, тогда надо делать дифф:
    - Например, хранить у себя дату последнего обновления по каждой сущности и при следующем запуске спрашивать только изменения.
    - Либо отслеживать версии источников: ChEMBL v33->v34, GtoPdb release x, etc.
    - Архитектура с pipeline modular позволяет реализовать обновление: просто перезапись данных. Но для сохранения истории (например, трендов) можно делать не truncate, а обновлять/вставлять измененные.
    - Если нужен **лог изменений**, можно расширить схему: timestamp columns, or a separate table for changes.

- **Поддержка дополнительных БД/форматов:** Можно подключить **Neo4j** или RDF-хранилище для графовых запросов. PostgreSQL же тоже можно обогащать, например, используя **JSONB** поля для хранения дополнительной информации (например, хранить полный JSON ответа источника в колонке для отладки – но это избыточно обычно).
- **API над агрегированными данными:** После интеграции может возникнуть потребность в предоставлении доступа к данным через API внутренним или внешним потребителям (например, чтобы не давать прямой доступ к БД).
    - Можно реализовать микросервис (на Flask/FastAPI) поверх нашей PostgreSQL, который по запросу выдает информацию: например, эндпоинт `/molecule/{id}/interactions` возвращает JSON с мишенями и публикациями.
    - Или GraphQL API, где клиент может запрашивать произвольные связывания (например: { molecule(name:"Aspirin"){ targets{ name, publications{ title } } } }).
    - Наша схема данных для этого достаточно понятна, GraphQL-сервер мог бы быть довольно прямым (есть библиотеки graphene, tartiflette etc.).
    - В рамках проекта это может быть **этапом 4 или отдельным проектом**, но закладывая расширение, можно:
    - Сразу писать код так, чтобы было легко повторно использовать модели и сессию (например, отделить logic извлечения из БД в функции, чтобы потом вызывать их из API).
    - Документацию подготовить с учетом внешних понятий (чтобы полям дать понятные описания – потом это пойдет в swagger).
- **Дополнительные сущности:** Возможен рост проекта: например, добавить сущность **Assay** (биологические эксперименты) из ChEMBL, чтобы хранить не только binary interaction (да/нет активность), а подробные данные экспериментов. Но это серьезно увеличит сложность (больше таблиц: assays, measurements, conditions). Если аналитика потребует – возможно, новый pipeline `chembl/assay` придется писать.
- **Интеграция с другими системами:**
    - Можно подумать о подключении **internal data** компании (если это в индустриальном контексте), например, собственные результаты исследований. Тогда архитектура ETL может быть расширена новыми pipelines, но общая структура позволяет это: просто новый provider (например, `internaldb`) с pipeline.
    - Если данные будут использоваться не только для чтения, но и возможно для *обогащения вручную* (например, эксперт хочет добавить заметку к взаимодействию), то нужно либо внести это в БД (доп колонки или таблицы для ручных аннотаций), либо иметь UI/API для этого. Пока это вне объема, но стоит понимать, что как только люди начнут активно пользоваться, появятся запросы «а можно поправить такую-то связь». Надо будет регулировать, чтобы не переписывали загруженные данные (так как при перезагрузке мы перетрем). Решение: отделить «агрегированные автоматически» и «ручные правки» в разные поля/таблицы.
- **Безопасность и доступ:**
- Если на базе будет поднят API, следить за безопасностью (ограничить кто может дернуть, если это чувствительно).
- Если база открыта для дашборда (например, подключена к Tableau/Metabase), то предусмотреть read-only пользователя.
- В коде не хранить секретные ключи (уже отмечалось).

Подводя итог, спроектированный ETL-пайплайн достаточно модульный и масштабируемый: можно добавлять новые источники данных, новые типы сущностей, увеличивать частоту

обновлений или оборачивать витрину собственным API – без кардинальных изменений архитектуры. Предусмотренное разбиение на слои и соответствие naming policy облегчит поддержку проекта, а продуманная стратегия нормализации и связывания данных обеспечит ценность полученной витрины для конечных аналитиков.

---

1 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

2 Accessing PubChem through PUG-REST - Part I - GitHub Pages
https://iupac.github.io/WFChemCookbook/datasources/pubchem_pugrest1.html

3 RESTful Interface - PubChem - NIH
https://pubchem.ncbi.nlm.nih.gov/docs/rdf-rest

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

5 OpenAlex API in Python — Scholarly API Cookbook - GitHub Pages
https://ua-libraries-research-data-services.github.io/UALIB_ScholarlyAPI_Cookbook/src/python/openalex.html

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

7 Welcome to crossref's documentation! — crossref 0.1.0 documentation
https://crossref.readthedocs.io/

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

9 Semantic Scholar - Awesome MCP Servers
https://mcpservers.org/servers/FujishigeTemma/semantic-scholar-mcp

10 Semantic Scholar Academic Graph API
https://www.semanticscholar.org/product/api

11 12 13 14 15 16 11-naming-policy.md
file://file-TtsHdFu2tmr4qCQ5ZWFnzd