ChatGPT

Файл: `model.md` **– Модель ABC-объектов (Assay, Activity, Target, TestItem)**

Модель ABC-объектов (Assay, Activity, Target, TestItem)

ABC-объекты – ключевые доменные сущности системы BioETL, включающие **ассай (Assay)**, **активность (Activity)**, **таргет (Target)** и **тестовый элемент (TestItem)**. Эти объекты представляют данные о биологическом эксперименте (ассай), его биологической цели (таргет), тестируемом веществе (тестовый элемент) и результате взаимодействия вещества с целью (активность). Ниже описано назначение каждого объекта, их основные атрибуты и типы, а также взаимосвязи между ними. Приведены диаграммы и таблицы для наглядности. Термины приводятся в соответствие с исходной документацией проекта BioETL.

Assay (ассай)

Назначение: Assay – это биологический эксперимент или тест, в рамках которого проверяется активность веществ. В контексте ChEMBL ассай представляет опубликованный эксперимент (например, энзимный анализ или клеточный тест), в котором измеряется эффект **TestItem** на определенный **Target**. Ассай содержит метаданные эксперимента: условия проведения, методику, тип измерения и т.д.

Атрибуты и типы: Структура данных ассая определяется схемой **AssaySchema** (Pandera DataFrameSchema). Обязательные поля включают: уникальный идентификатор ассая `assay_id` (ChEMBL ID эксперимента), бизнес-ключ `business_key` для уникальной идентификации, тип ассая `assay_type` (например, B – binding, F – functional), класс ассая `assay_class`, идентификатор связанного таргета `target_chembl_id`, а также другие поля с описанием категории, типа теста, ткани, документа и пр. [1] [2]. Ниже приведены основные поля сущности Assay:

Поле	Тип	Описание
`assay_id`	string	Идентификатор ассая (ChEMBL ID эксперимента)
`assay_type`	string	Тип ассая (например, B – связывание, F – функциональный)
`assay_class`	string	Класс ассая (категория/уровень эксперимента)
`target_chembl_id`	string	Идентификатор связанного таргета (ChEMBL ID биоцели)
`business_key`	string	Бизнес-ключ ассая для уникальной идентификации
...другие...	*различные*	Дополнительные атрибуты (описание, документ, ткани и т.п.)

Ассай наследует общие поля из базовой схемы (например, `chembl_release` – версия базы ChEMBL, `extracted_at` – дата извлечения, хеши строки) [3]. Бизнес-ключ обычно совпадает или составляется на основе внешнего ID (для ChEMBL это `assay_chembl_id`).

Связи: Один ассай может относиться к одному **таргету** (биологической цели), указанному в поле `target_chembl_id` [2], а также быть описан в одном **документе** (научной публикации, идентификатор `document_chembl_id` [4]). При этом один и тот же таргет или документ могут быть общими для многих ассайев. Ассай выступает «контекстом» для множества измерений **активности** – каждый ассай обычно включает несколько результатов активности для разных веществ. Таким образом, между Assay и Activity реализуется связь «один ко многим»: один ассай yield несколько активностей, описывающих эффекты разных тестовых веществ в данном эксперименте.

Пример: Например, ассай с идентификатором CHEMBL123456 описывает ферментативный тест, в котором измерялась ингибирующая активность веществ против фермента ACE. У этого ассая указан таргет CHEMBLACE (ангиотензин-превращающий фермент) и ссылка на публикацию. В ходе эксперимента протестированы несколько соединений (тестовых элементов), и для каждого зафиксирована численная активность (например, IC50). Эти результаты хранятся как объекты Activity, связаны с данным Assay.

Activity (активность)

Назначение: Activity – это результат измерения биоактивности: конкретное наблюдение или измеренное значение, полученное при тестировании **тестового элемента** в рамках конкретного **ассая**. Проще говоря, Activity представляет факт «вещество X проявило активность Y на таргет Z в эксперименте W». В ChEMBL под активностью обычно понимается единичное значение (например, IC50, Ki, процент ингибирования и т.п.) для комбинации «соединение–ассай».

Атрибуты и типы: Схема **ActivitySchema** определяет структуру данных активности. Ключевые поля: уникальный идентификатор активности `activity_id`, ссылка на эксперимент `assay_id` (связывает с Assay), ссылка на идентификатор биоцели `target_id` (таргет, может дублировать информацию из ассая), численное значение эффекта `value` (например, величина IC50) и единицы измерения `unit` [5]. Также присутствуют `business_key` – бизнес-идентификатор активности (как правило, составной ключ, комбинирующий идентификаторы ассая, вещества и прочие условия) – и поля-хеши `business_key_hash`, `row_hash` для контроля уникальности и целостности данных [6]. Ниже перечислены основные поля Activity:

Поле	Тип	Описание
`activity_id`	string	Идентификатор активности (уникальный ID результата)
`assay_id`	string	Идентификатор ассая, в рамках которого получен результат
`target_id`	string	Идентификатор таргета, связанного с активностью
`value`	float	Измеренное значение активности (числовой результат)
`unit`	string	Единица измерения активности (например, nM, %)

Поле	Тип	Описание
`business_key`	string	Бизнес-ключ активности (комбинация, уникально определяющая измерение)
...другие...	...	Дополнительные поля (например, флаги достоверности и т.д.)

Поле `target_id` обычно соответствует таргету, на который направлен ассай (из `target_chembl_id` ассая). Поле идентификатора тестового элемента (`molecule_chembl_id`) может не храниться напрямую в ActivitySchema (в зависимости от реализации пайплайна), но логически каждая Activity связана с определенным **TestItem** – веществом, чья активность измерена. Бизнес-ключ активности формируется так, чтобы однозначно идентифицировать запись активности (например, сочетание идентификаторов ассая и вещества, типа измерения и условия).

Связи: Объект Activity связывает **ассай** и **тестовый элемент**. Каждая Activity относится ровно к одному Assay (через `assay_id`) и характеризует один TestItem (вещество, участвовавшее в тесте). Таким образом, Activity является сущностью «многие ко многим» между Assay и TestItem: один ассай порождает много Activity (по числу протестированных веществ), и одно вещество может иметь много Activity (из различных ассайев). Кроме того, через ассай каждая Activity косвенно связана с определенным **таргетом** (таргетом того эксперимента) и с **документом** (публикацией, где описан эксперимент).

Пример: В рамках вышеупомянутого ассая по ACE каждая протестированная молекула имеет запись активности. Например, для соединения Aspirin (CHEMBL25, тестовый элемент) измерено значение IC50 = 50 μM против таргета ACE в ассаях типа ингибирования. Эта информация сохранится как объект Activity: со ссылкой на идентификатор ассая CHEMBL123456, значением 50, единицей “μM” и указанием тестового элемента Aspirin. Другой тестируемый компонент, например Captopril, будет иметь свою Activity (например, IC50 = 0.02 μM), и так далее. Каждая такая запись характеризует эффективность конкретного вещества в данном эксперименте.

Target (таргет)

Назначение: Target – это биологическая «цель», на которую направлено действие тестируемых веществ. Таргетом может быть макромолекула (например, белок-фермент, рецептор), клеточная линия, микроорганизм или иной объект исследования, чья реакция на вещество измеряется в ассаях. В системе BioETL **Target** представляет структурированные данные о такой биологической цели: название, тип (белок, комплекс, путь и т.д.), организм происхождения и другие характеристики.

Атрибуты и типы: Данные таргета структурированы с помощью схемы **TargetSchema**. Обязательные поля включают: уникальный идентификатор таргета `target_id` (например, ChEMBL Target ID), официальное название `pref_name` (предпочитаемое имя белка/гена или объекта), вид организма `organism` (например, *Homo sapiens*), тип таргета `target_type` (классификация – например, Single Protein, Protein Family, Tissue и др.), а также вычисляемые поля для обеспечения уникальности: `business_key` (бизнес-ключ таргета, возможно составленный из имени и организма), его хеш `business_key_hash` и хеш всей записи `row_hash` [7]. Ниже перечислены ключевые поля Target:

Поле	Тип	Описание
`target_id`	string	Идентификатор таргета (ChEMBL ID биоцели)
`pref_name`	string	Название таргета (например, имя белка)
`organism`	string	Организм, к которому относится таргет
`target_type`	string	Категория таргета (тип биологической сущности)
`business_key`	string	Бизнес-идентификатор таргета (для дедупликации)
`business_key_hash`	string	Хеш бизнес-ключа (для проверки уникальности)
`row_hash`	string	Хеш всей строки (для контроля целостности данных)

Таргет содержит атрибуты, позволяющие идентифицировать его однозначно и устранить дубли (например, два таргета с одним именем и организмом будут сведены к одному бизнес-ключу). Дополнительно могут храниться описания, синонимы, информация о последовательности (для белков) и другие детали, поступающие из ChEMBL или смежных источников.

Связи: Один таргет может фигурировать во многих **ассаях**. В данных ассая хранится ссылка на `target_chembl_id` – таким образом, множество ассайев могут ссылаться на один и тот же таргет (например, популярный белок-фермент может быть целью десятков различных экспериментов) [2]. Соответственно, через ассай таргет связан и с активностями: все **Activity**, относящиеся к ассаям данного таргета, являются проявлениями активности веществ на этот таргет. Прямой связи между Target и Activity в модели обычно нет (они связываются через Assay), но логически можно считать, что один таргет имеет множество связанных активностей (совокупность всех измерений по нему). Таргеты могут быть также связаны с внешними справочниками (например, Gene ID, Uniprot ID), но внутри BioETL эти связи представлены либо в атрибутах таргета, либо через отдельные сервисы.

Пример: Таргет с идентификатором CHEMBL3951 соответствует ферменту «Ангиотензин-превращающий фермент» (ACE) человека. Его тип – «Single Protein», организм – *Homo sapiens*. Этот таргет фигурирует в разных ассаях – например, в нашем примере он является целью ферментативного ингибиторного анализа. В результате, в системе будут многочисленные активности, связанные с CHEMBL3951: каждое измерение IC50 или % ингибирования разных веществ против ACE.

TestItem (тестовый элемент)

Назначение: TestItem – это тестируемое вещество (химическое соединение), чья активность измеряется в ассаях. Данная сущность соответствует молекулам из химической базы (в ChEMBL – это **Molecule**, запись о соединении с уникальным CHEMBL ID). В BioETL **TestItem** представляет основные свойства химического соединения, необходимые для идентификации и последующей обработки, а также служит для связи с результатами активностей. Проще говоря, тестовый элемент – это «кандидат» (лекарство, химическое соединение), который испытывается на активность.

Атрибуты и типы: Структура данных TestItem определяется схемой **TestItemSchema**. К обязательным полям относятся: идентификатор молекулы `molecule_chembl_id` (например,

CHEMBL ID вещества), ключ InChI `inchi_key` для однозначной идентификации структуры, канонический SMILES `canonical_smiles` (текстовое представление структуры), молекулярная масса `molecular_weight`, и ряд других свойств молекулы [8]. Как и у других сущностей, у тестового элемента может определяться бизнес-ключ (например, сам `molecule_chembl_id` может выступать бизнес-ключом) и хеши для контроля. Кроме того, TestItemPipeline обогащает данные, полученные из ChEMBL, дополнительной информацией из внешних источников (например, PubChem), поэтому в данных могут появляться поля, связанные с родительскими структурами, идентификаторами в PubChem и т.д. [9]. Ниже основные поля TestItem:

Поле	Тип	Описание
`molecule_chembl_id`	string	Идентификатор молекулы (ChEMBL ID соединения)
`inchi_key`	string	InChI-Key – уникальный ключ химической структуры
`canonical_smiles`	string	Канонический SMILES (строковое представление молекулы)
`molecular_weight`	float	Молекулярная масса соединения
`business_key`	string	Бизнес-ключ (например, тот же CHEMBL ID вещества)
...другие...	...	Дополнительные свойства (формула, LogP, источники и т.д.)

Схема TestItem обеспечивает валидацию полученных данных о молекулах – проверяется наличие идентификаторов, корректность форматов (например, валидность InChI ключа), типы данных и т.п. [10]. Возможна композиция схем: для полного набора молекул используется **TestitemsSchema** – расширенная схема для всего датасета молекул, включающая, например, колонки от обогащения (добавленные поля PubChem) [11].

Связи: Каждый TestItem (соединение) может участвовать в множестве **ассайев** и, соответственно, иметь множество записей **Activity**. В рамках одного ассая один тестовый элемент обычно дает одну или несколько активностей (например, несколько типов измерений). Таким образом, между TestItem и Activity связь «один ко многим»: одно вещество имеет много измерений активности (в разных экспериментах), каждая Activity относится к одному веществу. Прямой ссылки на ассай в данных TestItem нет – связь осуществляется через активности (или бизнес-ключи). Кроме того, тестовый элемент может быть связан с внешними базами данных (PubChem ID, патентами и т.д.), но эти связи отражаются в дополнительных полях или внешних справочниках.

Пример: Возьмем тестовый элемент Aspirin (CHEMBL25). Его данные: InChIKey=BSYNRYMUTXBXSQ-UHFFFAOYSA-N, формула C9H8O4, молекулярная масса ~180. Aspirin фигурирует в десятках ассайев ChEMBL – от ферментативных тестов (например, ингибирование COX1) до клеточных анализов. Каждое такое испытание создаст запись активности: например, в нашем примере Aspirin имел IC50 ~50 μM против ACE; в другом ассае (COX1 ингибирование) Aspirin мог показать активность 20% при определенной концентрации. Все эти активности, связанные с CHEMBL25, можно собрать, чтобы охарактеризовать профиль активности вещества.

Взаимосвязи объектов

На рисунке ниже приведена диаграмма отношений между рассмотренными объектами:

```
erDiagram
    DOCUMENT ||--o{ ASSAY : "описаны в"
    TARGET ||--o{ ASSAY : "используются в"
    ASSAY ||--o{ ACTIVITY : "порождает"
    TEST_ITEM ||--o{ ACTIVITY : "участвует в"
```

Диаграмма: взаимосвязи между объектами. Один документ содержит несколько ассайев; один таргет может быть целью многих ассайев; один ассай yield множество активностей; один тестовый элемент участвует во многих измерениях активности.

Основные связи можно также представить в таблице:

Объект	Связанные объекты	Тип связи
Assay	Target, Document, Activity	1 Assay – 1 Target; 1 Assay – 1 Document; 1 Assay – *N* Activity (многие результаты)
Activity	Assay, TestItem (через Assay), Target (через Assay)	*N* Activity – 1 Assay; *N* Activity – 1 TestItem; *N* Activity – 1 Target (via Assay)
Target	Assay, Activity (через Assay)	1 Target – *N* Assay; 1 Target – *N* Activity (косвенно)
TestItem	Activity, Assay (через Activity)	1 TestItem – *N* Activity; 1 TestItem – *N* Assay (через активности)
Document	Assay	1 Document – *N* Assay

Из таблицы видно, что **Assay** является центральной сущностью, соединяющей **Target** (био-цель) и **Document** (источник) с множественными **Activity**. **Activity** служит связкой между **TestItem** и **Assay**, фиксируя результат взаимодействия. **Target** и **TestItem** находятся как бы на «противоположных концах» эксперимента, и их связь реализуется через конкретные ассайы и активности.

Аналогии с реальными примерами

Чтобы лучше понять эти сущности, приведем аналогию из лабораторной практики. Представьте научную статью (Document), в которой исследователи сообщают об эксперименте (Assay) – например, они проверяют, как разные препараты снижают активность фермента ACE (Target). У них есть набор лекарственных соединений (TestItem), скажем, препараты для снижения давления. Каждый препарат протестирован в эксперименте, и измерено конкретное значение – например, концентрация, при которой достигается 50% ингибирование фермента (это и есть измеренная Activity для каждой пары «препарат–фермент»). В итоге статья содержит таблицу результатов: список веществ с их IC50 против ACE. В терминологии нашей системы: документ содержит ассай (тест ингибирования ACE), таргет – ACE, тестовые элементы – различные препараты, активности – значения IC50 для каждой пары. Такая

модель позволяет однозначно структурировать и хранить информацию о биоактивности соединений.

Источники: Использованы схемы данных и документация проекта BioETL [1] [7], а также конфигурация ChEMBL-пайплайнов [4] для подтверждения атрибутов и связей между объектами. Соглашения о бизнес-идентификаторах и наследовании базовых полей взяты из архитектурных принципов проекта [3].

Файл: `architecture.md` **– Архитектура системы (ETL и общая архитектура проекта)**

Архитектура проекта BioETL

Архитектура BioETL спроектирована для эффективной организации процессов **ETL** (Extract – Transform – Load) при интеграции данных из внешних источников биоактивности. В этой части документации изложены ключевые аспекты архитектуры: общая схема ETL-процесса, модульная структура системы, основные слои (доменный, инфраструктурный, интеграционный), важные компоненты и точки расширения. Архитектура следует принципам повторного использования и расширяемости, минимизируя дублирование кода и обеспечивая единообразие подходов во всех пайплайнах [12].

Общий обзор ETL-архитектуры

BioETL реализует конвейер обработки данных, состоящий из стандартных стадий: **извлечение (Extract)** данных из внешнего API, **преобразование (Transform)** и нормализация, **валидация (Validate)** структуры и качества данных, и **запись (Write)** артефактов результата [13] [14]. Эта последовательность этапов задаёт скелет каждого пайплайна. Ниже представлена упрощённая схема потока данных:

```
Внешний API (например, ChEMBL)
    ↓
Клиент API (SourceClient, например ChemblClient)
    ↓
Извлечение (Extractor, получает сырые данные)
    ↓
Парсинг (Parser, конвертирует JSON в DataFrame)
    ↓
Трансформация (Transformer, обогащает и нормализует данные)
    ↓
Нормализация (Normalizer, приведение типов, заполнение значений)
    ↓
Валидация (Validator/ValidationService, проверка по схеме)
    ↓
Запись (Writer, сохранение в файлы/базу)
```

```
    ↓
Артефакты (файлы Parquet, метаданные, отчёты качества)
```

Схема: высокоуровневый поток данных ETL. Каждый этап реализован отдельным компонентом, соответствующим интерфейсу абстрактного класса (ABC) для своей роли. Например, Extractor, Transformer, Writer – все реализуют общий контракт `StageABC`. Это упрощает замену или модификацию стадий без влияния на остальную систему.

Слои системы: Архитектура разделена на несколько условных слоёв:

- **Доменный слой (Domain):** содержит бизнес-логику ETL-процессов. Сюда входят сами **пайплайны** (Pipeline) для разных сущностей, включая их этапы, а также доменные модели данных (схемы, описание объектов Assay, Activity и др.). Доменный слой оперирует терминами предметной области (активности, таргеты, ассайы) и отвечает за корректное преобразование исходных биоданных в целевой формат. Например, класс `ChemblAssayPipeline` в доменном слое описывает, как именно извлечь и обработать данные ассая из API ChEMBL [15].

- **Инфраструктурный слой (Infrastructure):** предоставляет общие сервисы и утилиты, не зависящие от конкретной предметной области. К нему относятся компоненты для работы с HTTP (кеширование, лимитирование запросов, повторные попытки и т.д.), система логирования, конфигурация, управление файлами, а также сервисы валидации и записи результатов. Этот слой обеспечивает стандартизированные решения задач, необходимых всем пайплайнам. Например, **Unified API Client** инкапсулирует логику HTTP-взаимодействия (ретраи, тайм-ауты, разделение на страницы) для любых внешних API [16], а **SchemaRegistry** и **ValidationService** предоставляют механизм централизованной проверки данных по схемам [17] [18].

- **Интеграционный слой (Integration):** отвечает за взаимодействие с внешними системами и поставщиками данных. В него входят модули **клиентов внешних API** (например, ChemblClient, PubChemClient) и адаптеры, преобразующие внешние данные во внутренние структуры. Этот слой содержит реализацию интерфейсов доступа (на основе ABC) под конкретные веб-сервисы. Также здесь могут находиться интеграции с системами хранения (например, запись файлов на диске, S3 и пр.). В BioETL каждое внешнее API имеет свой клиент, реализующий общие абстракции: например, ChemblClient использует унифицированный HTTP-клиент и стандартные настройки для обращения к REST API ChEMBL.

Такое разделение по слоям делает архитектуру более чистой: доменный код не зависит от деталей HTTP или формата конфигурации, а инфраструктура не «знает» деталей биодоменной логики.

Модульная структура и компоненты

Проект организован модульно, с акцентом на повторное использование. **Основные модули** BioETL можно разделить по функциональности:

- **Core (ядро пайплайнов):** базовые классы пайплайнов и стадий. Здесь определён `PipelineBase` – абстрактный базовый класс, реализующий оркестрацию этапов ETL

(последовательный вызов extract → transform → validate → write) [19]. Он также управляет ресурсами (автоматически вызывает `dispose()` у клиентов), обеспечивает атомарность записи результатов и поддерживает режим имитации (dry-run) [19]. От него наследуются профильные базовые классы, например `ChemblBasePipeline`, добавляющие логику, общую для всех пайплайнов ChEMBL [20] – инициализацию сервисов извлечения/записи для ChEMBL, валидацию типичных параметров (batch_size и др.), а также предоставляющие **хуки** (набор методов для точек расширения: `pre_transform()`, `domain_enrich()` и т.п.) [20]. Конкретные пайплайны (ActivityPipeline, AssayPipeline, TargetPipeline и др.) наследуются от этих базовых и реализуют только уникальную бизнес-логику: особенности трансформации полей, специфичные проверки и т.д. [21]. Такая иерархия позволяет не дублировать общий код – например, типовая процедура экстракции и сохранения реализована один раз в базовом классе.

- **Abstract Base Classes (ABC) и интерфейсы:** Внутренние **ABC-интерфейсы** формализуют контракт для важных компонент. Например, `SourceClientABC` определяет интерфейс клиента для внешнего API, `PaginatorABC` – интерфейс пагинатора (постраничной загрузки), `TransformerABC` – для трансформации данных, `ValidatorABC` – для валидаторов, `WriterABC` – для модулей записи и т.д. [22]. Для каждого такого ABC есть реализация по умолчанию (Default/Impl), которую можно заменить или расширить при необходимости. Пример: `PaginatorABC` реализован как `OffsetPaginatorImpl` для последовательной загрузки страниц, `RetryPolicyABC` – как `ExponentialBackoffRetryImpl` (экспоненциальная повторная попытка), `RateLimiterABC` – как `TokenBucketRateLimiterImpl` и т.д. [22]. Эти абстракции позволяют менять стратегии (например, внедрить другой алгоритм повторных попыток) без переписывания логики пайплайна – достаточно указать альтернативную имплементацию в конфигурации или коде.

- **Clients (клиентский слой):** Модуль клиентов внешних API включает классы, которые знают, как обращаться к конкретным веб-сервисам (ChEMBL, PubChem, etc.). Они используют инфраструктурные возможности (HTTP-клиент, ретраи, лимиты) через унифицированный подход. В частности, **UnifiedAPIClient** предоставляет единый интерфейс для выполнения HTTP-запросов с соблюдением необходимых политик (повторы, лимиты, circuit breaker и пр.) [16]. Например, вместо прямого использования библиотеки `requests` с явным указанием таймаутов и обработкой ошибок, код пайплайна создаёт UnifiedAPIClient с нужной политикой и вызывает метод `fetch_one()` или `fetch_all()` для получения данных [23]. Это значительно упрощает код этапа Extract и гарантирует применение стандартных инфраструктурных практик (например, экспоненциальный backoff при падении соединения, ограничения на частоту запросов, автоматическое прерывание при длительных сбоях API).

- **Schemas и валидация:** Все сущности (Activity, Assay, Target, TestItem и др.) имеют определённые схемы данных (на базе Pandera), которые регистрируются в системе. **SchemaRegistry** хранит соответствие между именем сущности и классом схемы [24]. **ValidationService** на этапе *Validate* извлекает нужную схему и проверяет DataFrame на соответствие: наличие всех колонок, типы, ограничения, порядок колонок [25] [18]. Валидация централизована и обязательна для каждого пайплайна, что обеспечивает высокое качество и одинаковую структуру данных на выходе всех процессов. Например, перед записью результатов **ChEMBLActivitySchema** валидирует, что у Activity присутствуют все необходимые поля в нужном типе и порядке [25]. Сами схемы спроектированы с максимальным

переиспользованием: общие колонки (такие как `chembl_release`, timestamps, хеши) вынесены в базовые классы (например, BaseChemblSchema), а специфические схемы наследуют их [26]. Это позволяет при изменении требований (скажем, добавлении нового общего поля) поправить один базовый класс и автоматически распространить изменение на все сущности [27].

- **Сервис записи (Writer) и артефакты:** На завершающей стадии результаты сохраняются в унифицированном формате. **UnifiedOutputWriter** (реализация `WriterABC`) выполняет атомарную запись данных: пишет временный файл и затем переименовывает его в целевой для гарантии целостности [28]. Данные сохраняются преимущественно в формате Parquet (колоночный формат, эффективный для аналитики), а также генерируются вспомогательные файлы: **meta.yaml** с метаданными (версия пайплайна, версия источника данных, количество записей, контрольные суммы и пр.) [29], отчёты качества (например, CSV с показателями полноты или распределения данных) [30]. Структура выходных файлов стандартизирована: для каждого запуска создаётся отдельная директория с понятным именем (например, `activity_chembl_<дата>`), в ней – основной файл данных (parquet), YAML с метаданными и отчёты [31]. Планирование имён и путей выполняет компонент **ArtifactPlanner**. Такой подход упрощает последующую загрузку и анализ: все результаты самодокументированы и отделены друг от друга, а наличие метаинформации позволяет отследить происхождение данных.

Архитектурные принципы и снижение связности

Архитектура BioETL руководствуется рядом принципов, направленных на **минимизацию дублирования** и повышение гибкости:

- **Иерархия конфигураций:** Вместо копирования одинаковых параметров по всем пайплайнам, используется многоуровневая система YAML-конфигураций [32]. Определён базовый профиль (например, `chembl_default.yaml`) с общими настройками для всех ChEMBL-пайплайнов. Над ним надстраиваются профили окружения – development, production – которые переопределяют лишь отличия. Наконец, каждый конкретный пайплайн имеет свою конфигурацию с уникальными параметрами сущности (фильтры, поля) [33]. При запуске можно указать профиль (dev/prod) и при необходимости перекрыть отдельные значения параметров через флаг `--set` [34]. Этот подход устраняет дублирование и позволяет централизованно менять, например, URL API или параметры пагинации для всех пайплайнов разом.

- **Централизация общих функций в базовых классах:** Как отмечалось, все повторяющиеся части логики вынесены в `PipelineBase` и специализированные базовые классы. Конкретные пайплайны реализуют только различающуюся часть (свою бизнес-логику) [21]. Например, `ActivityPipeline` переопределяет лишь детали трансформации активностей, используя все остальное «из коробки» от `ChemblBasePipeline`. Аналогично, в компоненте нормализации данных создан базовый класс **BaseChemblNormalizer**, реализующий единый процесс приведения типов, очистки строк, вычисления хешей, установки значений по умолчанию [35]. Специфичные нормализаторы (для Activity, Assay, Target) добавляют только особую логику – например, парсинг специфических полей или дополнительные проверки форматов ID [36]. Это устраняет тройное дублирование кода нормализации для каждой сущности.

- **Повторное использование ABC и default-реализаций:** Перед написанием нового компонента разработчикам предписано проверить, не существует ли уже подходящего ABC с типовой реализацией [37]. Благодаря каталогу ABC (есть реестр со списком всех доступных абстракций и их стандартных реализаций [38]), можно быстро найти нужный функционал. Например, потребность в ограничении скорости запросов решается подключением готового `RateLimiterABC` с `TokenBucketRateLimiterImpl` вместо реализации с нуля [22]. Такая архитектурная политика ускоряет разработку новых пайплайнов и гарантирует единое поведение (все используют одни и те же механизмы ретраев, пагинации и т.п.).

- **Расширяемость через хуки и плагины:** Система предоставляет **точки расширения**: методы и события, которые можно переопределить для изменения поведения без форка базового кода. Примеры – методы `pre_transform`, `post_transform`, `pre_validate` в пайплайнах, позволяющие внедрить дополнительную обработку до или после стандартных шагов. Также предусмотрены механизмы расширения через регистрацию новых реализаций ABC (например, если нужно особое поведение, можно реализовать свой класс, унаследованный от StageABC, и подключить его). В проекте существует концепция PipelineHookABC – абстрактного хука для расширения функциональности пайплайна, который можно реализовать, не меняя код основного конвейера. Такая гибкость позволяет адаптировать систему под новые требования (новый формат данных, другой способ хранения, интеграция дополнительной аналитики) с минимальным вмешательством. Важно, что ввод новых компонентов происходит только при явной необходимости – архитектурное правило гласит «не создавать новый слой или модуль, если задачу можно решить расширением существующего» [39]. Перед добавлением чего-то принципиально нового разработчик проверяет чеклист: нельзя ли встроить это в core, реализовать как новую стадию или через уже имеющийся ABC? [40]. Это предотвращает избыточное усложнение системы.

- **Единообразие и прозрачность:** Все пайплайны реализованы по одному шаблону, что упрощает чтение и сопровождение кода. Соглашения по именованию и структуре (например, названия сущностей, полей данных – snake_case, одинаковые названия ключевых колонок вроде `business_key`, `hash_row`) строго соблюдаются [41]. Наличие подробной документации (как эта) и справочников терминов [42] гарантирует, что новые участники команды быстро разберутся в устройстве системы.

Ключевые технологии и фреймворки

BioETL построен на стеке Python, активно используя следующие библиотеки и инструменты:

- **Pandas** для работы с табличными данными (каждый этап оперирует `pd.DataFrame`).
- **Pandera** для декларативного описания схем DataFrame и валидации данных [43] [44]. Pandera позволяет задать ожидаемые колонки и типы, а затем автоматически проверять DataFrame – это основа компонента ValidationService.
- **Requests / HTTPX** (через UnifiedAPIClient) для вызовов REST API. Несмотря на то, что непосредственно `requests` не вызывается в коде пайплайнов, он используется внутри унифицированного клиента.
- **PyYAML** для чтения конфигураций (профили и pipeline config – это YAML-файлы).
- **Parquet (PyArrow)** для эффективного хранения результатов.

- **Logging (структурированное логирование)** – система логирования настроена на вывод структурированных сообщений (JSON-формат), чтобы облегчить анализ работы пайплайнов и отладку.

Кроме того, проект придерживается архитектурных стилей, таких как *ETL-конвейер*, *Domain-Driven Design* (в части четкого разделения доменных сущностей и сервисов), *Dependency Injection* через передачу конфигураций и использование ABC (реализации внедряются через конструкторы или параметры, а не жестко создаются внутри). Это делает систему более тестируемой и настраиваемой.

Источники: Архитектурные решения и принципы обобщены на основе внутренних документов проекта [45] [37], а также диаграммы потока данных ETL [13]. Информация о ключевых компонентах взята из референсной документации (Pipeline Base, UnifiedAPIClient, SchemaRegistry) [19] [16]. Структура слоёв и модулей – из обзора архитектуры проекта BioETL и правил проектирования [46].

Файл: `development.md` **– Разработка и процессы (API, соглашения, ETL-процессы)**

Разработка: API, соглашения и процессы ETL

В этом разделе описаны практические аспекты разработки и использования системы BioETL: каким образом предоставляется API для запуска и расширения пайплайнов, какие приняты соглашения по форматам данных и идентификаторам, как протекают процессы ETL (пошагово), и как организовано взаимодействие с внешними REST API. Эти сведения помогут понять, как пользоваться существующими компонентами и как добавлять новые, соблюдая требования проекта.

API системы и структурированные интерфейсы

Запуск и управление пайплайнами: BioETL предоставляет интерфейс командной строки (CLI) для запуска ETL-конвейеров. Главная команда – `bioetl run <pipeline_name>` – позволяет выполнить определенный пайплайн (например, `bioetl run activity_chembl` запустит процесс загрузки активностей из ChEMBL). Через флаги CLI можно указать профиль конфигурации (`--profile development` или `--profile production`) и переопределить отдельные настройки (`--set key=value`) [34]. Такой CLI выступает основным API системы для пользователя-разработчика: вместо написания кода достаточно вызвать готовую команду.

Конфигурация пайплайнов: Каждый пайплайн описывается YAML-конфигурацией (находится в `configs/pipelines/`). В ней указываются параметры: URL или endpoint внешнего API, список полей для загрузки, фильтры (например, по датам или идентификаторам), настройки пакетирования запросов и пр. [47] [48]. Есть также общие профили (`configs/profiles/`) для разных окружений. Конфигурации поддерживают наследование (через ключ `extends`) – это реализует принцип, что большая часть настроек берется из базовых профилей, а специфичные параметры – в файле конкретного пайплайна [33]. API конфигурации структурирован: все параметры имеют человекочитаемые имена, сгруппированы по функционалу (например, `client.timeout`, `pagination.limit`, `fields:` список полей). Разработчикам следует придерживаться существующих схем конфигов, добавляя новые поля только при необходимости и документируя их в README по конфигурации [49].

Интерфейсы классов (ABC): Для расширения функционала без ломки существующего кода используется механизм **абстрактных базовых классов (ABC)**. Например, если необходимо добавить новый тип проверки качества данных, можно реализовать класс, наследующий `QualityMetricABC`, и зарегистрировать его. В конфигурации или коде пайплайна предусмотрены точки подключения таких расширений. Структурированные интерфейсы (ABC) гарантируют, что новая реализация будет совместима с оркестратором пайплайна. В проекте ведется каталог всех ABC-интерфейсов (документ **ABC Index**), где перечислены их методы и предназначение [50]. Перед разработкой нового компонента рекомендуется ознакомиться с этим списком, чтобы либо воспользоваться готовым Default-решением, либо правильно встроить свою реализацию.

Программный доступ: Хотя основной сценарий – запуск через CLI, BioETL также может использоваться программно. Например, можно импортировать класс пайплайна и вызвать его методы из кода Python. Все пайплайны имеют унифицированный интерфейс: они инициализируются конфигурацией и имеют метод `run()` или эквивалент, запускающий последовательность стадий. Также присутствуют вспомогательные API, например, `validate_config()` для проверки корректности настроек, `list_pipelines()` для получения списка доступных конвейеров и т.д. Такой API важен для интеграции BioETL в другие системы (например, запуск ETL из веб-приложения или по расписанию).

Конвенции и соглашения проекта

Проект строго придерживается **единых соглашений** по форматам данных, именованию и структурам, что облегчает совместимость компонентов:

- **Форматы идентификаторов:** Все внешние идентификаторы (ChEMBL ID для различных сущностей) сохраняются как строки определенного формата. Например, идентификаторы соединений, ассайев, документов начинаются с префикса `CHEMBL` и цифр – шаблон проверяется регулярным выражением `^CHEMBL\d+$` [3]. Если запись не соответствует ожидаемому формату, она отбраковывается или помечается как некорректная на этапе нормализации. Внутренние бизнес-ключи формируются либо на основе этих ID, либо как их комбинация (например, для Activity бизнес-ключ может быть конкатенацией `assay_id|molecule_id`). Все бизнес-ключи и хеши представляются строками фиксированной длины (например, `hash_row` – 64-символьный хеш SHA256) [51].

- **Наименование полей и файлов:** Принят стиль snake_case (строчные буквы с подчеркиванием) для названий всех колонок в данных. Это относится и к JSON-полям, получаемым из API: если внешний источник использует другой стиль, он приводится к snake_case на этапе парсинга. Имена CSV/Parquet файлов строятся по шаблону `<entity>_<source>.parquet` (например, `assay_chembl.parquet`), отчеты – `quality_report_<entity>_<source>.csv`. Такие имена также продублированы в meta.yaml для удобства навигации [31].

- **Требования к структурам данных:** Перед записью каждого набора данных гарантируется строгий порядок колонок и наличие всех обязательных полей. Валидация с параметром `ordered=True` требует, чтобы порядок соответствовал определению схемы [52]. Например, даже если DataFrame по содержанию верен, но колонки переставлены местами,

ValidationService выстроит их в правильном порядке. Это необходимо для *детерминизма*: чтобы хеши и сравнения наборов данных разных запусков совпадали при идентичном содержимом [53]. Пустые значения (`None`) заполняются оговорёнными дефолтами (например, пустые числовые поля – NaN, строки – пустая строка или специальный маркер), чтобы не нарушать типы. Для каждой сущности определено, какие поля могут быть nullable, а какие нет (например, `assay_id` не может быть NULL, `confidence_score` может) [54]. Эти правила зафиксированы в схемах и автоматически применяются.

- **Документация и комментарии:** Каждая новая функция или класс должны иметь docstring на английском или русском (в проекте используется двуязычный стиль, но ключевые понятия часто оставляются на английском, чтобы совпадать с названиями в коде). Комментарии сопровождают сложные участки алгоритмов, особенно когда происходит преобразование данных или используются хитрые костыли для обхода ограничений API. Соглашение: **не допускать «магических чисел» и непонятных действий без пояснения** – все параметры, вроде лимитов, адресов, вынесены в конфиг или константы с осмысленным именем, а не раскиданы по коду [55].

- **Контроль качества данных:** На уровне разработки приняты меры для обеспечения качества: наряду с основными артефактами, генерируются **отчеты качества (QC)**, упомянутые выше. Разработчик, добавляющий новый пайплайн, должен определить метрики качества (сколько пустых значений, диапазоны, распределения) и включить их расчет. Эти отчеты помогают валидировать данные не только структурно, но и по содержанию (например, увидеть, что для 95% записей указан единый таргет – подозрение на дублирование). Также действует правило: **не игнорировать ошибки в данных** – если API вернул некорректное значение (например, текст вместо числа), пайплайн либо преобразует его к корректному типу, либо явно логирует проблему. В логах в случае ошибок схемы указывается, какая колонка не прошла проверку, с примерами значений.

Процесс ETL: шаги и преобразования

Каждый пайплайн проходит через четыре главных стадии. Рассмотрим их подробнее на примере процесса загрузки активности (Activity) из ChEMBL:

1. **Extract (Извлечение):** На этом шаге компонент **Extractor** соединяется с внешним API и вытягивает сырые данные. Он строит **запросы** согласно `ExtractionDescriptor` (специальный объект, содержащий критерии выгрузки – например, диапазон ID, фильтры по видам активностей) [56]. Экстрактор использует клиент API – например, `ChemblClient` – который уже включает в себя политику повторных попыток и лимитирования запросов. Если данных много, задействуется пагинация: через `PaginatorABC` определяется, как перебрать все страницы (ChEMBL, например, позволяет запрашивать 20–100 записей за раз, потому Paginator формирует последовательность запросов с нужными OFFSET) [57]. Extractor собирает все полученные ответы (в формате JSON) и преобразует их в DataFrame – обычно с помощью **Parser**: класс Parser берет JSON из API и маппирует поля в колонки DataFrame по заданной схеме сопоставления. На выходе этапа Extract мы имеем **сырые несформатированные данные** – таблицу, близкую к тому, что отдает внешний сервис [58]. В дополнение, собирается метаданные выгрузки: сколько записей получено, за сколько времени, были ли ошибки/повторы и т.д. Этап Extract обязан обеспечить устойчивость к сетевым ошибкам: при

временных проблемах он повторяет запросы (за счет RetryPolicy), при недоступности сервиса может задействовать Circuit Breaker, чтобы не блокировать процесс надолго [59]. Если часть запросов не удалась вовсе, это логируется, но процесс старается вытащить максимум доступных данных (пропуская только то, что совсем не удалось загрузить).

2. **Transform (Преобразование):** Компонент **Transformer** получает DataFrame с сырыми данными и приводит их к внутренней канонической структуре. Первым делом обычно идет **обогащение доменными полями** – добавление колонок, которые нужны для дальнейшей работы, но отсутствуют во входных данных. Например, добавляется колонка `chembl_release` (версия базы ChEMBL, может браться из конфигурации или ответа API) [60]. Далее применяется **нормализация данных**: в BioETL это выделено в отдельный под-этап. Для каждого поля определена спецификация нормализации (**ColumnNormalizationSpec**): что сделать – сконвертировать тип (строку в число или дату), обрезать лишние пробелы, привести даты к UTC, заменить null на дефолт и т.д. [35]. Все эти операции выполняются Normalizer'ом – для ChEMBL сущностей используется `BaseChemblNormalizer` плюс специфический нормализатор. Например, ActivityNormalizer дополнительно парсит поле ligand_efficiency, если оно присутствует [36]. Transformer вызывает Normalizer для каждой колонки (либо применяет пакетно). После нормализации выполняется **выравнивание структуры** под целевую схему: убеждаются, что все ожидаемые колонки присутствуют, и если каких-то не хватает – добавляются с дефолтными значениями; если колонки в неправильном порядке – упорядочиваются как в схеме [61]. Результатом этапа Transform является очищенный и обогащенный DataFrame, готовый к проверке.

3. **Validate (Валидация):** За эту стадию отвечает **ValidationService**, использующий Pandera-схемы. Он берет DataFrame после трансформации и соответствующую ему схему из **SchemaRegistry** по названию сущности (например, для активности – ChEMBLActivitySchema) [62]. Далее вызывается метод `schema.validate(df)` с параметрами strict=True, ordered=True [63]. Это означает, что DataFrame не должен содержать лишних колонок, должен содержать все требуемые, и типы/значения должны соответствовать определённым условиям (например, `activity_id` – строка непустая, `value` – число, `molecule_chembl_id` – строка, соответствующая регэкспу CHEMBL ID). При обнаружении несоответствия бросается исключение SchemaError, которое фиксируется: обычно pipeline логирует ошибку и прекращает работу, если данные критически не соответствуют схеме. В некоторых случаях допускается **пустой DataFrame** (например, если по заданным критериям не было данных) – тогда ValidationService сформирует пустой DataFrame с правильными колонками, чтобы на выходе всё равно получить нужную структуру [64]. Валидация – критический этап, защищающий от попадания в хранилище «грязных» данных. Он гарантирует, что после завершения пайплайна все наборы данных стандартизированы. В процессе валидации также может быть включена дополнительная логика: например, проверка бизнес-правил (отсутствие дублей по `business_key`, диапазоны значений показателей лежат в разумных пределах и пр.). Эти проверки, если нужны, добавляются в схему Pandera (например, ограничение на форматы значений через `pa.Field(regex=...)` или специальные Check-функции).

4. **Write (Запись):** Последний этап – **запись результатов** – выполняется компонентом **Writer**. Он получает финальный DataFrame, готовый к сохранению, и информацию о том, куда писать (пути к файлам, сгенерированные ArtifactPlanner, и уникальный префикс запуска `run_id`) [65]. Запись в BioETL сделана атомарной и детерминированной: Writer сначала сортирует

DataFrame по бизнес-ключам (чтобы при повторных запусках файлы Parquet имели один и тот же порядок строк) [66]. Затем данные записываются во временный файл формата Parquet. Используется метод `write_dataset_atomic()` – он сохраняет файл с временным именем, а затем переименовывает (MOVE) в целевое имя [67]. Благодаря этому, если во время сохранения что-то пойдет не так, старые результаты не перезапишутся испорченными данными (атомарность). После успешной записи генерируются дополнительные файлы: файл `meta.yaml` с метаданными пайплайна – версия, количество записей, время выполнения, хеш-суммы файлов и т.д. [29]. Также, если включен режим расчета метрик качества, Writer сохранит **QC отчеты** – например, `quality_report_activity_chembl.csv` (в нем могут быть статистики пропущенных значений, распределения результатов) и `correlation_report_activity_chembl.csv` (если предусмотрен анализ корреляций между колонками) [30]. После завершения стадии Write на диске (или ином хранилище) появляется папка с именем, соответствующим запускаемому пайплайну и времени запуска, содержащая все артефакты. Пайплайн возвращает объект **WriteResult**, в котором прописаны пути к сохраненным файлам и сводная информация (например, объем данных).

Таким образом, процесс ETL обеспечивает полный цикл от извлечения исходных сведений до готовых для анализа структурированных данных, сопровождаемых метаданными. Каждый этап изолирован и протестирован, что упрощает отладку: ошибки при сетевом вызове ловятся на Extract, проблемы с типами – на Validate, и т.д.

Взаимодействие с внешним REST API

BioETL специально спроектирован для удобной интеграции с REST API источников данных. Рассмотрим, как организована работа с внешним API на примере ChEMBL:

- **Единая точка входа – клиент:** Для каждого внешнего сервиса реализован свой клиент, но все они опираются на **унифицированный HTTP-стек**. В случае ChEMBL это `ChemblClient`, который, в свою очередь, использует `UnifiedAPIClient` из инфраструктуры. Разработчику пайплайна не нужно вручную вызывать `requests.get` или разбираться с заголовками – достаточно сформировать объект запроса (URL + параметры) и передать его UnifiedAPIClient. Этот клиент автоматически применит политику ограничений: например, ChEMBL API имеет ограничение по количеству запросов в секунду, поэтому при частых вызовах RateLimiter приостановит отправку до истечения интервала [16]. Если API временно не отвечает (ошибка 503 или таймаут), включается **RetryPolicy** – повторит запрос несколько раз с экспоненциальной задержкой [16]. Если же сервис лежит продолжительное время, **CircuitBreaker** остановит дальнейшие попытки и выдаст ошибку, чтобы не зависнуть навечно [16]. Все эти механизмы конфигурируются: например, число попыток, базовая задержка, максимальная скорость запросов – заданы в профиле и могут быть изменены без правки кода.

- **Pagination и Batch-запросы:** Многие REST API (включая ChEMBL) не отдают сразу все данные по одному запросу, а требуют постраничного обхода или позволяют запрашивать блоками. BioETL поддерживает оба варианта. В конфигурации указывается тип пагинации – например, `pagination: offset` с размером страницы 500 – и Extractor автоматически будет вызывать API метод, увеличивая offset от 0, 500, 1000, … пока не получит все записи. Если API позволяет передавать список ID для выборки (batch), то используется **BatchPlan**: например, ChEMBL для

активности позволяет запрашивать 25 ID за раз – тогда формируется план, разбивающий полный список требуемых ID на группы по 25 и делающий серию запросов [68]. Эта логика скрыта внутри PaginatorABC/Extractor; разработчику достаточно указать `batch_size: 25` и список ID, а система сама распределит их по запросам.

- **Парсинг ответов JSON:** Получив ответ от API (формат JSON или XML), Extractor передает его парсеру – обычно это метод или отдельный класс **Parser**. В конфигурации описан **mapping** полей: какие поля JSON соответствуют каким колонкам DataFrame. Например, поле JSON `"assay_chembl_id"` маппится в колонку `assay_id`, `"standard_value"` -> `value`, и т.п. Парсер проходит по каждому записанному объекту в JSON и формирует строку DataFrame в соответствии с этим маппингом. При необходимости он приводит типы (числовые строки -> float и т.д.) и раскладывает вложенные структуры. В некоторых случаях API возвращает вложенные списки/объекты – парсер либо нормализует их сразу (например, превращает список значений в строку, разделенную точкой с запятой), либо сохраняет как есть в JSON-колонке для дальнейшей обработки в Transform. В BioETL маппинг полей для каждого пайплайна задан в YAML-конфиге (секция `fields:`) [47], а универсальный Parser читает эту схему и применяет. Такой подход позволяет легко добавить/убрать поля, просто отредактировав конфиг, без изменения кода парсера.

- **Ограничения и фильтрация на стороне API:** Часто возникает потребность получать не все данные, а отфильтрованные (например, все активности после определенной даты, или только связанные с конкретным таргетом). BioETL позволяет передавать параметры фильтрации в API-запрос. В конфигурации они могут указываться, например: `filters: target_chembl_id=CHEMBL3951` – тогда клиент будет вызывать URL вроде `https://api.chembl.org/.../activity?target_chembl_id=CHEMBL3951`. Другой пример – указание полей, которые должен вернуть API (если есть возможность селективного выбора полей, чтобы снизить объем ответа). Эти опции задаются в конфиге и автоматически подкладываются при формировании запросов. Разработчику важно убедиться, что поддерживаемые фильтры соответствуют документации API – для этого в репозитории ведутся заметки по возможным параметрам (в Confluence или markdown есть раздел **API справочники**). Если API не поддерживает нужную фильтрацию, отброс не нужных данных выполняется уже после загрузки, на стадии Transform (например, можно в `pre_transform` отфильтровать DataFrame по нужному условию).

- **Обработка ошибок API:** Помимо сетевых ошибок, могут быть ошибки уровня бизнес-логики: API вернул код ошибки (400/404) – например, запрошен несуществующий ресурс, или слишком большой offset. Клиент API должен корректно обработать такие случаи. В BioETL типовой подход: при 4XX ошибке (кроме 429 Too Many Requests) – считать, что данных нет или запрос некорректен, и пропустить/залогировать; при 429 или 5XX – повторять с задержкой (эти коды покрывает RetryPolicy). Также, если API возвращает не-JSON (HTML с сообщением об ошибке), клиент это обнаружит и сгенерирует исключение, которое отловит Extractor. В логах любая неожиданная ситуация отмечается, но пайплайн старается идти до конца, собирая всё, что возможно.

В итоге, разработчику, работающему с REST API через BioETL, предоставляется высокоуровневый инструмент: достаточно настроить конфигурацию и, при необходимости, реализовать небольшие кастомные шаги в Transform, а низкоуровневое взаимодействие, повторные вызовы и соблюдение

ограничений берёт на себя архитектура. Это ускоряет разработку новых интеграций и снижает количество ошибок, связанных с невнимательным обращением к внешним сервисам.

Источники: Описание шагов ETL основано на детализации из документа **Data Flow** (Architectural Overview) [14] [69]. Логика работы с REST API подтверждена фрагментами кода и документации: принципами UnifiedAPIClient [23] [16], а также настройками конфигурации ChEMBL (поля, фильтры) [47]. Информация о соглашениях (форматы ID, схемы) взята из правил проекта [3] и референсов схем в Pandera [7].

---

[1] [43] 04-assay-schema.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/02-pipelines/schemas/04-assay-schema.md

[2] [4] [47] [48] [54] assay.yaml
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/configs/pipelines/chembl/assay.yaml

[3] [12] [16] [19] [20] [21] [22] [23] [26] [27] [32] [33] [34] [35] [36] [37] [38] [39] [40] [45] [51] [55] 04-architecture-and-duplication-reduction.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/project/04-architecture-and-duplication-reduction.md

[5] [6] activity_schema.py
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/src/bioetl/core/schemas/activity_schema.py

[7] [44] 03-target-schema.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/02-pipelines/schemas/03-target-schema.md

[8] [10] 01-testitem-schema.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/02-pipelines/schemas/01-testitem-schema.md

[9] [11] INDEX.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/02-pipelines/chembl/testitem/INDEX.md

[13] [14] [28] [29] [30] [31] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] data-flow.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/architecture/data-flow.md

[15] 00-assay-chembl-overview.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/02-pipelines/chembl/assay/00-assay-chembl-overview.md

[17] [18] [24] [25] [52] [53] 12-activity-chembl-schema.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/02-pipelines/chembl/activity/12-activity-chembl-schema.md

[41] [49] [50] README.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/README.md

42 46 index.md

https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/overview/index.md