ChatGPT

Архитектура ETL-проекта интеграции данных из научных API

1. Структура файлов проекта

Проект организован в виде Python-пакета `bioetl` внутри директории `src`. Ниже приведена структура файлов с указанием назначения основных пакетов:

```
src/
├── bioetl/
│   ├── core/             # базовые классы для пайплайнов ETL и runtime-
контекст
│   ├── pipelines/        # реализация пайплайнов и их оркестрация (ChEMBL,
PubMed и др.)
│   ├── clients/          # API-клиенты для внешних источников, HTTP-
адаптеры, протоколы обмена
│   ├── io/               # ввод-вывод данных, абстракции хранилищ и
сериализации результатов
│   ├── config/           # загрузка конфигураций и управление настройками
проекта
│   ├── domain/           # описания схем данных, валидация (например, с
помощью Pandera), нормализация данных
│   ├── utils/            # вспомогательные утилиты и общие функции
│   └── cli/              # CLI-команды (на базе Typer) для запуска/
управления пайплайнами
└── tests/
    └── bioetl/
        ├── pipelines/    # тесты для пайплайнов
        ├── clients/      # тесты для клиентского слоя
        └── config/       # тесты для конфигураций
```

Каждый подпакет содержит модули, соответствующие своему слою ответственности в ETL-процессе. Например, в `core` хранятся абстракции и инфраструктурные классы пайплайна, в `pipelines` реализованы конкретные пайплайны для различных источников (используя общие базовые классы из core), в `clients` — классы для взаимодействия с внешними REST API (клиенты), а в `io` — логика сохранения результатов (запись на диск, в базу и пр.). Тесты отражают ту же структуру, проверяя ключевые компоненты каждого слоя.

2. Базовые абстрактные классы и их интерфейсы

Архитектура проекта основана на наборе абстрактных базовых классов (ABC) и протоколов, которые определяют общий контракт для реализации ETL-пайплайна и сопутствующих компонентов. Ниже приведён полный список таких базовых абстракций (только интерфейсы, без конкретных реализаций), необходимых для построения нового пайплайна:

Пайплайн и оркестрация ETL

- **PipelineRuntimeBase** – базовый класс runtime для пайплайна, отвечающий за общую оркестрацию этапов ETL. Определяет, например, абстрактный метод `build_stage_plan(context, options) -> tuple[StageDescriptor, ...]` для формирования детерминированной последовательности стадий выполнения [1]. Наследники этого класса задают, какие этапы (extract, transform, load и др.) будет выполнять конкретный пайплайн.
- **PipelineBase** – абстрактный базовый класс конкретного ETL-пайплайна (наследуется от *PipelineRuntimeBase*), в котором непосредственно описывается логика этапов. Включает абстрактные методы `extract(descriptor, options) -> DataFrame` (извлечение данных из источника) и `transform(df, options) -> DataFrame` (преобразование/очистка данных) [2]. Таким образом, любой новый пайплайн должен реализовать эти методы. Дополнительно базовый класс уже реализует типовые шаги: опциональный `validate` (валидация результатов, например с Pandera) и `save_results` (сохранение трансформированных данных с помощью сервиса записи).

Клиентский слой доступа к данным

- **BaseClient** – абстрактный базовый класс для высокоуровневых клиентских классов, через которые пайплайн получает данные из внешнего API. Задает интерфейс с методами: `fetch_one(request) -> Record` (получить одну запись по запросу), `iter_records(request) -> Iterator[Record]` (итератор по записям на основе запроса), `iter_pages(request) -> Iterator[Page]` (итератор по страницам результатов) и др., а также метод `close()` для освобождения ресурсов [3] [4]. Конкретные реализации (напр. клиент ChEMBL или PubMed) должны переопределить эти методы, инкапсулируя логику вызовов API.
- **SourceClientABC** – интерфейс низкоуровневого клиента для непосредственного общения с внешним источником данных (транспортный уровень). Содержит один ключевой метод `send(request) -> response`, который выполняет запрос (HTTP/REST) к внешнему сервису и возвращает ответ [5]. Это позволяет отделить формирование запроса и обработку ответа от собственно выполнения HTTP-запроса.
- **RequestBuilderABC** – абстракция построителя запросов к API. Определяет методы для формирования запросов: `build_initial() -> Request` (создание начального запроса без указания курсора/страницы) и `build_for_page(cursor) -> Request` (создание запроса для следующей страницы на основе курсора или индекса) [6]. Реализация для конкретного API знает, как сформировать правильные параметры (фильтры, идентификаторы, лимиты и т.д.) для вызова данного API.
- **ResponseParserABC** – интерфейс парсера ответов от API. Определяет методы `parse_items(response) -> Iterable[ParsedItem]` для извлечения элементов данных (записей) из «сытого» ответа и `extract_cursor(response) -> Optional[str]` для получения курсора (например, идентификатора следующей страницы или next-token) из ответа [7]. Реализация зависит от формата ответа конкретного API (JSON, XML) и структуры в нём данных и метаданных пагинации.
- **PaginatorABC** – абстракция стратегии постраничного обхода (пагинации). Содержит метод `get_next_request(prev_request, last_response) -> Optional[Request]`, который по последнему выполненному запросу и полученному ответу определяет, есть ли следующая страница, и если да, формирует следующий запрос [8]. Например, реализация может извлекать из ответа параметр «next page» и модифицировать запрос (смещение, номер страницы или курсор) для продолжения загрузки данных. Если данных больше нет, возвращается `None`.

- **CacheABC** – абстрактный кэш для результатов запросов. Определяет методы `get(key) -> Optional[value]`, `set(key, value)` и `invalidate(key)` для чтения, записи и инвалидации кэш-записей [9]. В контексте ETL-пайплайна кэш может использоваться, например, чтобы не делать повторный запрос к API для уже загруженных данных (по ключу запроса) или для хранения промежуточных результатов обработки.

Обработка данных и вывод результатов

- **DeduplicatorABC** – абстракция компонента очистки данных от дубликатов. Предоставляет метод `deduplicate(records, key_fn) -> Iterable[Record]`, который на основе функции определения бизнес-ключа `key_fn` фильтрует повторяющиеся записи [10]. Реализация этого интерфейса позволяет убрать дубли из набора данных (например, если одно и то же наблюдение пришло дважды из разных источников или повторилось при пагинации).
- **WriterABC** – базовый интерфейс для записи итоговых данных пайплайна в хранилище. Содержит метод `write(records, output_path) -> None`, который отвечает за сохранение коллекции записей (обычно Pandas DataFrame или итерируемых объектов) в заданное место (файл, каталог, базу данных и т.д.) [11]. Разные реализации могут писать в CSV/JSON файлы, базы данных или облачные хранилища, но через единый метод интерфейса.
- **MetadataWriterABC** – упрощённый интерфейс для сохранения метаданных о запуске пайплайна рядом с основными результатами. Содержит метод `write_metadata(target_path, metadata) -> None` для записи служебных метаданных (например, JSON с информацией о времени выполнения, параметрах запуска, версии источника данных) по указанному пути [12]. Это позволяет при реализации пайплайна легко добавлять сохранение дополнительной информации о выполнении без смешивания с основными данными.

Логирование, обработка ошибок и интеграция с CLI

- **LoggerAdapterABC** – абстрактный класс адаптера логгера, определяющий интерфейс структурированного логирования в рамках пайплайна. Включает методы `info(message, **fields)`, `warning(message, **fields)` и `error(message, **fields)` для логирования событий разного уровня с возможностью добавления полей контекста (например, идентификатор запуска, имя пайплайна и др.) [13]. Реализация может оборачивать стандартный logging или внешнюю библиотеку логирования, обеспечивая единый формат логов.
- **ErrorPolicyABC** – интерфейс стратегии обработки ошибок во время выполнения пайплайна. Содержит метод `decide(exc: Exception, context: Mapping[str, Any]) -> ErrorAction`, который на основе возникшего исключения и контекста выполнения возвращает некое решение (например, константу *ErrorAction*, указывающую продолжить выполнение, пропустить проблемный блок или завершить пайплайн с ошибкой) [14]. Таким образом, конкретная политика ошибок (продолжать при незначительных ошибках, или останавливать сразу) может быть реализована отдельно и подставлена в пайплайн.
- **CLICommandABC** – абстрактный класс для команд CLI, связанных с пайплайнами. Определяет обязательные элементы CLI-команды: свойство `name: str` (имя/алиас команды) и метод `run(args: list[str]) -> int` для выполнения команды [15]. Это позволяет расширять CLI новыми командами, например, для запуска определённого пайплайна, генерации отчётов, очистки данных и пр., следуя единообразному интерфейсу. Команды на основе этого ABC регистрируются в пакете `bioetl.cli` и автоматически подхватываются Typer для формирования CLI-интерфейса.

Каждая из перечисленных базовых абстракций задаёт четкий контракт (набор методов), который должны реализовать конкретные классы. Благодаря этому, проект достигает модульности: компоненты пайплайна (источники, обработчики, хранение результатов, и т.д.) можно разрабатывать и изменять независимо, соблюдая интерфейсы. Новый ETL-пайплайн строится путем наследования от **PipelineBase** (реализация логики `extract/transform`) и комбинирования клиентских классов (**BaseClient** + связанные **RequestBuilder/ResponseParser/ Paginator**) и классов вывода данных (**WriterABC** и др.) по необходимости, что соответствует принципам чистой архитектуры ETL.

---

[1] runtime.py
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/src/bioetl/core/pipeline/runtime.py

[2] unified.py
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/src/bioetl/core/pipeline/unified.py

[3] [4] client_abc.py
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/src/bioetl/clients/base/client_abc.py

[5] SourceClientABC.md
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/_docs/SourceClientABC.md

[6] RequestBuilderABC.md
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/_docs/RequestBuilderABC.md

[7] ResponseParserABC.md
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/_docs/ResponseParserABC.md

[8] PaginatorABC.md
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/_docs/PaginatorABC.md

[9] CacheABC.md
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/_docs/CacheABC.md

[10] DeduplicatorABC.md
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/_docs/DeduplicatorABC.md

[11] WriterABC.md
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/_docs/WriterABC.md

[12] MetadataWriterABC.md
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/_docs/MetadataWriterABC.md

[13] LoggerAdapterABC.md
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/_docs/LoggerAdapterABC.md

14 ErrorPolicyABC.md

https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/_docs/ErrorPolicyABC.md

15 CLICommandABC.md

https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/_docs/CLICommandABC.md