ChatGPT

Шаблон базового ETL-пайплайна для BioETL

Этот документ описывает минимальную реализацию общей логики ETL-пайплайна на базе существующих абстрактных классов проекта **BioETL**. Шаблон предназначен для ускорения разработки новых пайплайнов: он предоставляет основу, от которой можно наследоваться и реализовывать конкретные Extract-Transform-Load процессы.

Обзор и назначение шаблона

Шаблон представляет собой Python-модуль (один файл `.py`), содержащий базовый класс пайплайна. Этот класс наследуется от абстрактного `PipelineBase` проекта BioETL и реализует требуемые методы, а также включает встроенную поддержку типовых возможностей:

- **Структура ETL (Extract → Transform → Validate → Write):** шаблон определяет абстрактные методы `extract`, `transform`, `validate`, `write` (выполняет сохранение) и реализует их заглушки (например, `pass` или выбрасывание `NotImplementedError`). Это даёт разработчику чёткий скелет этапов пайплайна.
- **Оркестрация выполнения:** базовый класс содержит метод `run()`, который управляет последовательным вызовом стадий ETL и сбором результатов. Он также обрабатывает режим *dry run*, то есть прогон без сохранения результатов.
- **Интеграция с CLI:** класс поддерживает интерфейс командной строки через интеграцию с `CLICommandABC`. Благодаря этому каждый пайплайн можно зарегистрировать как отдельную CLI-команду для запуска.
- **Логирование:** в конструкторе и в ходе выполнения инициализируется логирование (например, с помощью `structlog` либо стандартного `logging`). Логирование связывается с уникальным идентификатором запуска (`run_id`) и именем пайплайна, что упрощает отладку и анализ выполнения.
- **Валидация данных (Pandera):** на этапе `validate` предусмотрена поддержка схем Pandera. Если в пайплайне определена схема (`DataFrameSchema`), она будет применена для валидации DataFrame. В противном случае валидация пропускается с соответствующим сообщением в логах.
- **Управление ресурсами:** шаблон реализует методы `register_client` и `close_resources` для безопасной работы с внешними ресурсами (например, подключения к базам данных, HTTP-клиенты). Все зарегистрированные клиенты автоматически закрываются по окончании работы пайплайна (в блоке `finally` метода `run`).
- **Гибкость через внедрение зависимостей:** конструктор класса принимает дополнительные объекты-компоненты (реализации ABC-интерфейсов) для опционального изменения поведения: `SideInputProviderABC` (сайд-инпуты для обогащения данных), `DeduplicatorABC` (удаление дубликатов), `ValidatorABC` (дополнительная пользовательская валидация), `WriterABC` и `MetadataWriterABC` (кастомная запись результатов и метаданных), `PathStrategyABC` (стратегия формирования путей для выходных файлов) и другие. Эти зависимости по умолчанию не задействованы, но могут быть переданы при инициализации и использованы в соответствующих стадиях.

Шаблон представлен в двух частях: - **Документация (вы читаете её):** файл `docs/guides/pipeline_template.md` описывает возможности шаблона, его использование и примеры. - **Код шаблона:** файл `src/bioetl/core/pipeline_base_template.py` содержит реализацию класса-шаблона с подробными комментариями и docstring на русском языке.

Ниже приводятся ключевые элементы шаблона и рекомендации по его использованию.

Содержимое шаблона и реализация методов

В шаблоне определён класс `PipelineBaseTemplate`, наследующий `PipelineBase`. Он реализует абстрактные методы и предоставляет базовую (минимальную) логику, которую можно переопределить в наследниках:

- **Конструктор** `__init__`**:** принимает объект конфигурации пайплайна (`PipelineConfig`) и `run_id` (строка или `None`). В конструкторе происходит инициализация базовых свойств:
- Вызов `super().__init__(config, run_id)` для установки конфигурации и инициализации логгера, идентификатора запуска и др. базовых вещей в `PipelineBase`.
- Сохранение зависимостей (если переданы) в атрибуты: `side_input_provider`, `deduplicator`, `validator`, `writer`, `metadata_writer`, `path_strategy` и др. Эти компоненты соответствуют абстрактным интерфейсам из архитектуры (например, `SideInputProviderABC`, `DeduplicatorABC` и т.д.) и могут использоваться на стадиях обработки.
- Установка человекочитаемого имени пайплайна `pipeline_name`. По умолчанию шаблон берёт имя класса, но в целом это может задаваться в конфигурации.

- (Опционально) Инициализация логгера, если базовый класс этого не сделал. Например, привязка `pipeline_name` и `run_id` к структурированному логеру. Шаблон ориентируется на то, что `PipelineBase` уже настроил логирование, поэтому дополнительная настройка выполняется только при необходимости.

- **Метод** `extract(*args, **kwargs) -> pd.DataFrame`**:** абстрактный метод извлечения данных. В шаблоне он определён с `raise NotImplementedError`, то есть требует обязательной реализации в конкретном пайплайне. На этапе extract происходит подключение к внешним источникам данных (БД, API, файлы) и загрузка данных в pandas DataFrame. Согласно требованиям, метод `extract` **не должен** выполнять запись на диск, а лишь возвращает сырые данные.

- **Метод** `transform(df: pd.DataFrame) -> pd.DataFrame`**:** абстрактный метод трансформации данных. В шаблоне также заглушка (`raise NotImplementedError`). На этом этапе предполагается реализация бизнес-логики: очистка, нормализация данных, обогащение справочной информацией, дедупликация, агрегация и прочие преобразования с **исключением внешнего I/O**. Входом служит DataFrame, полученный из `extract`, выходом — преобразованный DataFrame для последующей валидации и записи.

Шаблон *может* использовать внедрённые зависимости на этом этапе: - Если задан `side_input_provider` (реализует `SideInputProviderABC`), можно через него получить дополнительные справочные данные и объединить с основным DataFrame. - Если задан `deduplicator` (реализует `DeduplicatorABC`), его метод `deduplicate` может быть вызван

для удаления дублей. Например, шаблон может продемонстрировать вызов `self.deduplicator.deduplicate(df.itertuples(), key_fn)` с подходящей функцией ключа. - Другие компоненты уровня трансформации (например, `BusinessKeyDeriverABC`, `LookupEnricherABC`, `TransformerABC` и т.п.) также могут применяться, если интегрированы.

В базовом шаблоне эти детали не реализованы (оставлены для конкретных пайплайнов), но при наличии переданных зависимостей разработчик может задействовать их внутри своего `transform`.

- **Метод** `validate(df: pd.DataFrame) -> pd.DataFrame`**:** базовая реализация проверки качества данных. В шаблоне метод присутствует (не абстрактный), поэтому наследник *может* его переопределить, но не обязан — если устроит стандартное поведение:
- **Проверка наличия схемы.** Если у пайплайна определена схема валидации (например, атрибут или свойство `validation_schema` типа `pa.DataFrameSchema` от Pandera), то она будет применена. В шаблоне можно предусмотреть свойство `self.validation_schema` (по умолчанию `None`). Если схема присутствует, шаблон логирует начало валидации и применяет `self.validation_schema.validate(df)` [1] [2]. При успешной проверке возвращается исходный или скорректированный DataFrame.
- **Отсутствие схемы.** Если схема не задана, шаблон логирует предупреждение о пропуске валидации [3] и возвращает входной DataFrame без изменений.
- **Hash-колонки (опционально).** Если требуется детерминированная идентификация строк (например, контроль дубликатов при последующем объединении), можно добавить вычисление хэш-значений по определённым столбцам. Шаблон включает защищённый метод `_add_hash_columns(df)` в качестве места для этой логики. По умолчанию он может ничего не делать или добавлять хэш всей строки. Наследники могут переопределить `_add_hash_columns` для изменения набора колонок или метода хэширования.

Помимо Pandera, возможна интеграция дополнительного валидатора: если передан компонент `validator` (реализует `ValidatorABC`), шаблон мог бы вызвать его для дополнительной проверки. Например, `self.validator.validate_records(df)` для построчной проверки или агрегатных правил, если такой метод определён. В базовой реализации такой вызов можно закомментировать как подсказку.

- **Метод** `write(df: pd.DataFrame, output_path: Path) -> WriteResult`**:** базовая реализация сохранения результатов и записи метаданных. В шаблоне этот метод присутствует (не абстрактный), поэтому можно не переопределять, если поведение подходит, или переопределить для специфики (например, другой формат или дополнительный вывод).

В базовой реализации учитываются следующие аспекты: - **Формирование пути и формата.** Шаблон проверяет, является ли `output_path` директорией или файлом: - Если передана директория, по умолчанию генерируется имя файла, например `<pipeline_name>_<date>.csv` или `.parquet` в зависимости от требований (можно использовать текущее число или `run_id` для уникальности). - Если `output_path` содержит имя файла с расширением, используется оно. Формат определяется по расширению: `.csv` для CSV, `.parquet` / `.pq` для Parquet [4]. Если расширение не поддерживается, генерируется ошибка. - Опционально: если передана стратегия путей `path_strategy` (реализует `PathStrategyABC`), можно доверить ей формирование конечного пути. Например, шаблон может вызывать `output_path = self.path_strategy.get_output_path(output_path)` чтобы получить детализированный путь (с учётом партиционирования, имени набора данных и т.д.). - **Сортировка данных.** Перед

записью DataFrame может быть отсортирован по определённым колонкам для детерминированности вывода. Шаблон может проверить наличие атрибута `required_sort_columns` (список колонок) или метода `get_sort_columns()`. Если определено, производится сортировка DataFrame по этим колонкам [5]. Если нет, порядок строк остаётся как есть. - **Атомарная запись.** Для надёжности шаблон осуществляет запись через временный файл: 1. Создаётся целевая директория (`output_path.parent`) при необходимости [6]. 2. Определяется временное имя (например, добавляется суффикс `.tmp` к имени файла). 3. DataFrame записывается в этот временный файл (метод pandas: `DataFrame.to_csv` или `to_parquet`). 4. После успешной записи временный файл переименовывается в основной, заменяя предыдущий результат атомарно [7]. В случае сбоя старый результат не затрагивается. - **Запись метаданных (`meta.yaml`).** Шаблон генерирует YAML-файл метаданных рядом с данными [8]. Метаданные содержат: - `run_id` запуска; - имя пайплайна; - путь к результатному файлу; - формат данных; - количество записанных строк; - временные метки начала и окончания стадии записи; - длительность стадии; - флаг `dry_run` (для `write` всегда `False`, т.к. при dry run запись не выполняется); - информацию о хэш-колонках или других применённых подходах (если актуально).

```
Метаданные сохраняются в `<имя_файла>_meta.yaml`. В шаблоне можно
использовать модуль `yaml` для сериализации словаря метаданных.
```

- **Возврат результата.** Метод возвращает объект `WriteResult` (датакласс, описывающий результаты записи) [9]. Он включает как минимум:
  - Путь к итоговому файлу данных;
  - Путь к файлу метаданных;
  - Формат файла (`csv` или `parquet`);
  - Число записанных строк;
  - Длительность операции сохранения.

Шаблон реализует `WriteResult` через импорт из `bioetl.core.types` (или создаёт простой `dataclass WriteResult`).

Использование внедряемых зависимостей: Если в конструктор передан `writer` (`WriterABC`), шаблон может делегировать запись ему: например, вызвать `self.writer.write(df, output_path)` вместо самостоятельной записи. Это удобно, если у проекта есть стандартные механизмы атомарной записи. Аналогично, при наличии `metadata_writer` (`MetadataWriterABC`) можно доверить генерацию или сохранение мета-информации этому компоненту. В базовой реализации, однако, предполагается собственная простая запись, а переданные writer/metadata_writer можно задействовать по усмотрению (например, добавить комментарий или условный блок if).

- **Метод** `run(output_path: Path, *args, dry_run: bool = False, **kwargs) -> RunResult`**:** реализует полный запуск пайплайна (оркестратор). В базовом классе `PipelineBase` этот метод уже определён, но шаблон может предоставить собственную реализацию или вызывать `super().run(...)` с добавлением своего контекста. В шаблоне `PipelineBaseTemplate` метод `run` может быть реализован примерно так:
- **Логирование начала**: фиксируется время старта, пишется в лог сообщение о запуске пайплайна с указанием `run_id` и `output_path`.

- **Выполнение стадий**: последовательно вызываются `extract(*args, **kwargs)`, `transform(df)`, `validate(df)` [10]. Количество строк (если применимо) и длительность каждой стадии измеряются. Эти метрики можно собрать в словарь.
- **Dry run режим**: если `dry_run=True`, шаблон **пропускает** вызов `write` [11]. В логах отмечается, что сохранение пропущено (например: `"dry_run=True, write stage skipped"`). Переменная `write_result` устанавливается в `None`. Кроме того, можно на этапе dry run сгенерировать только `meta.yaml` с минимальной информацией (как будто набор пустой) — это по желанию.
- **Обычный режим**: если `dry_run=False`, вызывается `write(df, output_path)` и сохраняется результат записи в `write_result`.
- **Формирование** `RunResult`: по окончании собирается объект `RunResult` (тоже dataclass, импортируется из `bioetl.core.types`). Он содержит:
  - `run_id` запуска;
  - имя пайплайна;
  - флаг dry_run;
  - словарь метрик времени по стадиям (например, `{"extract_sec": ..., "transform_sec": ..., "validate_sec": ..., "write_sec": ..., "total_sec": ...}`);
  - количество строк после каждой стадии (например, `{"extracted_rows": N1, "transformed_rows": N2, "validated_rows": N3}`) – эти показатели можно заполнять если легко получить размерность DataFrame;
  - объект `write_result` (или None, если dry_run).
- **Закрытие ресурсов**: метод `run` обязан вызывать `close_resources()` в блоке `finally` [12]. Шаблон включает конструкцию `try: ... finally: self.close_resources()`, чтобы гарантировать освобождение ресурсов даже при исключениях.
- **Обработка ошибок**: при возникновении исключения на любой стадии, шаблон логирует ошибку (с указанием стадии, см. ниже про логирование) и повторно выбрасывает исключение выше, чтобы вызывающий код/CLI мог это обработать [12].

Если родительский класс `PipelineBase.run` уже содержит часть этой логики (что вероятно), шаблон может просто переопределять его для расширения (например, измерение метрик) либо вовсе не переопределять, полагаясь на базовую реализацию.

- **Логирование в ходе выполнения:** Шаблон ориентирован на структурированное логирование с контекстом. При старте и окончании каждой стадии можно вызывать `self.logger.info()` с полями: `stage="extract"` (или иное название стадии), `duration_sec=...`, `row_count=...` [13]. Также логируются особые события:
- Пропуск стадии (например, сообщение `"Skipping write stage (dry_run mode)"` при dry_run [14]).
- Отсутствие схемы (предупреждение `"No validation schema provided, skipping validate step"`).
- Исключения (с помощью `self.logger.exception` для вывода стектрейса).

Логер, инициализированный с `pipeline_name` и `run_id`, позволяет легко фильтровать записи по конкретному запуску. В шаблоне показано использование `structlog` (если доступен) или стандартного `logging` в подобном стиле.

- **Управление ресурсами:**
- Метод `register_client(name: str, client: Any) -> None`: добавляет внешний ресурс (клиент БД, API и т.п.) во внутренний реестр `_clients`. Шаблон предполагает, что

в базовом классе уже есть структура данных (например, словарь) для хранения клиентов. При регистрации можно логировать действие и убедиться, что имя уникально.
- Метод `close_resources() -> None`: проходит по всем зарегистрированным клиентам и вызывает у каждого `close()` или `dispose()`, если такие методы существуют [15]. Любые возникающие при закрытии ошибки логируются как предупреждения, но не прерывают процесс закрытия остальных. После закрытия всех соединений реестр очищается. Шаблон демонстрирует эту логику. Этот метод вызывается автоматически в `finally` блока `run` [12], поэтому вручную обычно его вызывать не нужно, разве что в специфических сценариях.

Данные методы обеспечивают безопасное завершение работы с любыми внешними ресурсами, которые могли быть инициализированы в `extract` или `transform` (например, сессии БД). Разработчик, используя шаблон, должен не забывать регистрировать такие ресурсы через `register_client`, чтобы не держать соединения открытыми по завершении работы.

Использование шаблона для создания новых пайплайнов

Чтобы создать новый ETL-пайплайн на основе данного шаблона, рекомендуется выполнить следующие шаги:

1. **Создайте класс пайплайна, наследуя от** `PipelineBase` (или от специализированного базового класса, если такой имеется для вашего источника данных). Вы можете начать с копирования содержимого `PipelineBaseTemplate` и переименования класса под ваш случай. Например, для загрузки данных ChEMBL с сущностью "Compound":

```
from bioetl.core.pipeline_base import PipelineBase

class CompoundChemblPipeline(PipelineBase):
    \"\"\"Пайплайн выгрузки и обработки данных о компонентах из ChEMBL.\"\"\"
    def extract(self, *args, **kwargs) -> pd.DataFrame:
        # Реализация извлечения данных (вызовы к API ChEMBL, формирование
DataFrame)
        ...

    def transform(self, df: pd.DataFrame) -> pd.DataFrame:
        # Реализация трансформации (очистка, обогащение, дедупликация)
        ...
        return df

    # Метод validate можно не переопределять, если достаточно базовой
проверки схемы.
    # Если нужна особая логика – например, агрегатные проверки – можно
переопределить.

    # Метод write тоже можно использовать базовый или переопределить для
специфичной записи.
```

Обратите внимание: в примере класс `CompoundChemblPipeline` унаследован прямо от `PipelineBase`. Это допустимо, так как шаблон лишь отражает ту же структуру. В некоторых

случаях может быть уместно наследоваться от промежуточного класса (например, `ChemblCommonPipeline`), если проект это предоставляет.

1. **Реализуйте метод** `extract`: подключитесь к источнику данных и верните `pandas.DataFrame`. Готовый DataFrame передаётся дальше по конвейеру.

2. **Реализуйте метод** `transform`: опишите все требуемые преобразования. Можно пользоваться утилитами и провайдерами:

3. Если вам нужны справочные данные для обогащения, подключите `SideInputProviderABC` и используйте его внутри `transform`.
4. Для удаления дублей можно воспользоваться `DeduplicatorABC` (если бизнес-ключ сложный, сначала реализуйте `BusinessKeyDeriverABC`).
5. Все внешние источники, к которым вы обращаетесь в `transform` (например, кэш или доп. сервисы), регистрируйте через `self.register_client` (если они открывают соединения).

6. После всех преобразований возвращайте итоговый DataFrame, готовый к валидации.

7. **Определите схему данных (опционально)**: если для набора данных существует явная структура, задайте Pandera-схему и привяжите её к пайплайну. Это можно сделать несколькими способами:

8. Добавить атрибут класса, например `validation_schema = MyEntitySchema` (где `MyEntitySchema` – ранее определённый `pa.DataFrameSchema`).

9. Либо переопределить метод `validate`, извлекая схему из конфигурации (`self.config`) или другого источника. Если схема определена, базовый `validate` автоматически проверит DataFrame. При расхождении фактических данных с ожидаемой структурой будет поднято исключение Pandera, которое нужно обработать (например, CLI обрабатывает такие ошибки отдельным кодом выхода).

10. **Уточните поведение** `write` **(при необходимости)**: По умолчанию `write` сохранит данные в CSV или Parquet файл и создаст `meta.yaml`. Если нужно иное:

11. Поменять формат или способ записи (например, писать в базу данных) – проще переопределить `write` полностью.
12. Для расширения, можно воспользоваться `WriterABC`: передайте реализацию в конструктор и внутри `write` вызовите её.

13. Если требуются дополнительные артефакты (например, quality report, manifest), их генерацию можно добавить после основной записи.

14. **Регистрация пайплайна в CLI**: Чтобы ваш пайплайн можно было запустить через командную строку, его нужно зарегистрировать. В проекте BioETL доступно несколько способов:

15. **Через реестр** `cli_registry`: открыть файл `src/bioetl/cli/cli_registry.py` и добавить вызов `register_pipeline("имя", класс)` с вашим новым классом. Имя регистрируемого пайплайна обычно включает источник, например

`"compound_chembl"` [16]. После этого команда запуска сформируется автоматически (например, `bioetl run-chembl-compound`).

16. **Через реализацию CLICommandABC**: Можно создать отдельный класс команды CLI, связанный с пайплайном. Шаблонный класс уже наследует `CLICommandABC`, что позволяет ему непосредственно выступать как команда. Он реализует свойство `name` (возвращает название команды) и метод `run(args)` для запуска. В простейшем случае `PipelineBaseTemplate.run()` может использоваться внутри `CLICommandABC.run`. Например:

```
class CompoundChemblPipeline(PipelineBaseTemplate, CLICommandABC):
    @property
    def name(self) -> str:
        return "compound_chembl"

    def run(self, args: list[str]) -> int:
        # Парсинг аргументов можно делегировать Typer'у или реализовать
самостоятельно.
        output = Path(args[0]) if args else Path("data/output/
compound.csv")
        try:
            self.run(output_path=output)  # Запуск пайплайна
            return 0  # успешное выполнение
        except Exception as e:
            self.logger.error(f"Pipeline failed: {e}")
            return 1  # сигнал ошибки
```

Однако, в существующей архитектуре удобнее пользоваться готовой функцией `create_pipeline_command` с Typer (см. документацию CLI BioETL). Поэтому, чаще достаточно зарегистрировать через `register_pipeline`, как упомянуто выше.

После реализации и регистрации, ваш пайплайн можно запускать через CLI утилиту проекта (например, `bioetl run-chembl-compound --output data/output/compound.parquet`). Он автоматически получит на вход конфигурацию, создаст `run_id` и выполнит все стадии согласно шаблону.

Полный код шаблона

Ниже приведён полный код файла `pipeline_base_template.py`. Он включает классы и методы, обсуждённые выше, со всеми необходимыми комментариями и документацией:

```
{% raw %}
{{#include ../../src/bioetl/core/pipeline_base_template.py}}
{% endraw %}
```

(Примечание: Код включён автоматически для синхронизации с актуальной версией шаблона. Не редактируйте блок выше напрямую, изменения следует вносить в `pipeline_base_template.py`.)

Заключение

Данный шаблон облегчает создание новых ETL-пайплайнов в BioETL, обеспечивая единообразный каркас и реализуя за вас общие моменты (логирование, dry run, метаданные, закрытие ресурсов). Разработчику остаётся сконцентрироваться на бизнес-логике конкретных стадий и особенностях источников/приёмников данных.

При использовании шаблона соблюдайте правила проекта: - Документируйте ваши классы и методы (все docstring на русском языке, как в шаблоне). - Следуйте соглашениям по именованию (например, имя класса пайплайна должно отражать сущность и источник, конфигурационные файлы — в каталоге `configs/pipelines/...` и т.д.). - Реализуйте и регистрируйте новые абстрактные компоненты (ABC), если встроенных недостаточно, но избегайте дублирования уже имеющихся решений (см. руководство **«Adding a New ABC»**). - Пишите тесты для новых пайплайнов: минимально unit-тесты на трансформацию и интеграционные на полный запуск.

Воспользовавшись шаблоном, вы получите работающий каркас, соответствующий архитектуре BioETL, и сократите время разработки нового функционала. Удачи в реализации новых пайплайнов!

Ссылки на документацию и код:

- Описание базового класса PipelineBase – архитектура и обязанности базового пайплайна BioETL [17] [18].
- Описание этапов пайплайна и взаимодействия компонентов – подробное описание, как Extract/Transform/Validate/Write соотносятся с разными абстракциями (Source, Processing, Validation, Output) [10] [19].
- Руководство: Добавление нового пайплайна – общий чеклист шагов при создании нового пайплайна (конфигурация, документация, тесты) [20].
- Модуль с регистрацией CLI-команд пайплайнов: `bioetl/cli/cli_registry.py` (регистрация классов для CLI) [16] и `bioetl/cli/cli_command.py` (механизм создания команд CLI на основе Typer) [21].

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [11] [12] [13] [14] [15] [17] [18] PipelineBase_description.md
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/_docs/PipelineBase_description.md

[10] [19] pipeline_objects_interaction.md
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/_docs/pipeline_objects_interaction.md

[16] cli_registry.py
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/src/bioetl/cli/cli_registry.py

[20] adding-new-pipeline.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/guides/adding-new-pipeline.md

[21] cli_command.py
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/src/bioetl/cli/cli_command.py