

Абстрактный класс PipelineBase для ETL-пайплайна

PipelineBase – это базовый абстрактный класс для табличных ETL-пайплайнов в проекте BioETL. Он реализует общий шаблон **Extract → Transform → Validate → Save** (извлечение → преобразование → валидация → сохранение) и обеспечивает единый интерфейс для всех пайплайнов (activity, assay, testitem, target, document). Класс берет на себя оркестрацию стадий, ведение логов, валидацию данных и атомарную запись результатов, что позволяет минимизировать дублирование кода в конкретных пайплайнах ¹. Ниже приводится структура класса PipelineBase и пояснения к его основным методам и свойствам, а также рекомендации по расширению этого класса для специфичных сущностей (например, ChEMBL-пайплайны).

Структура PipelineBase и ключевые компоненты

PipelineBase реализует шаблон ETL через следующие стадии:

- **Extract** – извлечение сырых данных (реализуется в подклассах).
- **Transform** – преобразование и очистка данных (реализуется в подклассах, с общими шагами, заданными в базовых классах).
- **Validate** – валидация структуры и качества данных (реализовано в базовом классе через схемы).
- **Save** – сохранение результатов на диск или в хранилище (реализовано в базовом классе, обеспечивает атомарность и метаинформацию).

Кроме основных стадий, PipelineBase поддерживает **dry-run режим** (выполнение без сохранения), централизованное управление ресурсами (например, закрытие подключений к внешним сервисам) и интеграцию с логированием ².

Класс ориентирован на обработку табличных данных с использованием **pandas DataFrame** на всех этапах конвейера. Это значит, что и исходные данные, и трансформированные, и валидированные – все представляются в виде DataFrame, что упрощает унификацию обработки.

Основные свойства PipelineBase (атрибуты, инициализируемые в конструкторе или при настройке пайплайна):

- `config` : Конфигурация пайплайна (например, словарь или объект), загружаемая из YAML-файлов профилей/настроек. В конфигурации задаются параметры источника данных (endpoint API, параметры фильтрации, размеры батчей), целевая сущность (`entity_name`), ключевые поля (`primary_key`), схема данных (описание колонок) и т.д. ³ ⁴. PipelineBase может применять проверку корректности конфигурации (например, через `_validate_common_config` в специализированных классах) ⁵.
- `logger` : Логгер (обычно реализует `LoggerAdapterABC`), используемый для структурированного логирования хода пайплайна (начало/окончание стадий, количество обработанных записей, ошибки и пр.) ².

- `schema_provider` / `validator`: Компонент для проверки данных. `PipelineBase` абстрагируется от конкретной реализации схемы и валидации через **SchemaProviderABC** и **ValidatorABC**⁶. То есть, схема данных может быть описана, например, на Pandera или Pydantic, но `PipelineBase` будет работать через унифицированный интерфейс. В большинстве случаев для табличных данных используется Pandera-схема (`DataFrameModel`) из реестра схем, но благодаря `SchemaProviderABC` можно подставить и другую реализацию (Pydantic-модель, JSON Schema и т.д.)⁶. Схема хранит структуру столбцов, их типы и ограничения, единообразно для всех пайплайнов.
- `write_service`: Компонент, реализующий запись результатов (реализация интерфейса **WriterABC**). По умолчанию в проекте используется `UnifiedOutputWriter`, обеспечивающий сохранение `DataFrame` в файл (например, Parquet/CSV) с атомарной заменой файла и генерацией сопутствующих артефактов (метаинформация, отчеты)⁷
⁸. Если `write_service` не настроен, попытка сохранить результаты вызовет ошибку⁹.
- (В специализированных подклассах могут добавляться и другие свойства, например, `extraction_service` или `api_client` для подключения к внешнему API, но `PipelineBase` определяет только общий интерфейс и не привязан к конкретному источнику.)

Ниже приведён каркас класса `PipelineBase` с основными методами (в виде Python-подобного псевдокода), за которыми следуют подробные пояснения:

```
from abc import ABC, abstractmethod
import pandas as pd

class PipelineBase(ABC):
    def __init__(self, config: dict, **dependencies):
        self.config = config
    # Конфигурация пайплайна (из YAML и профилей)
        self.logger = ...                      # Логгер для структурированного
    логирования
        self.schema_provider = ...
    # SchemaProviderABC для получения схемы валидации
        self.write_service = ...                # WriterABC для сохранения
    результатов
        self._resources = []                   # Список внешних ресурсов/
    клиентов для закрытия

    @abstractmethod
    def extract(self, descriptor: any, options: StageExecutionOptions) ->
pd.DataFrame:
        """Извлечение данных из внешнего источника (реализуется в
наследниках)."""
        pass

    @abstractmethod
    def transform(self, df: pd.DataFrame, options: StageExecutionOptions) ->
pd.DataFrame:
        """Преобразование данных (реализуется в наследниках, может
использовать общие шаги)."""

```

```

    pass

    def validate(self, df: pd.DataFrame, options: StageExecutionOptions) ->
        pd.DataFrame:
        """Валидация данных по схеме (использует SchemaProvider/Validator,
        опционально)."""

    # Если валидация отключена или схема не задана, просто вернуть df без
    изменений
        # Иначе выполнить проверку df по схеме (например, Pandera)
        return df

    def save_results(self, df: pd.DataFrame, artifacts: WriteArtifacts,
        options: StageExecutionOptions) -> WriteResult:
        """Сохранение результатов с атомарной записью и метаданными."""
        # Отсортировать df по ключевым колонкам для детерминированности
        # Применить хэширование или другие финальные преобразования, если
        нужно
        # Вызвать write_service.write_dataset_atomic(df, artifacts, ...)
        # для записи
        # Генерировать meta.yaml и другие артефакты
        return WriteResult(...)

    def prepare_run(self, options: StageExecutionOptions) -> None:
        """Хук, выполняющийся перед запуском пайплайна (по умолчанию ничего
        не делает)."""
        return None

    def finalize_run(self, run_result: RunResult) -> None:
        """Хук, выполняющийся после завершения пайплайна (по умолчанию ничего не
        делает)."""
        # Например: закрытие внешних подключений или клиентов
        return None

    def build_stage_plan(self, context: StageContext, options:
        StageExecutionOptions) -> tuple[StageDescriptor, ...]:
        """Формирование плана стадий (extract->transform->validate->save) с
        учетом настроек."""
        # Построить кортеж (pipeline) стадий в нужном порядке
        # Убрать validate, если нет валидатора; убрать save, если dry_run
        return stage_plan

```

(Примечание: `StageExecutionOptions`, `WriteArtifacts`, `WriteResult`, `StageContext`, `StageDescriptor` – это вспомогательные типы/классы из инфраструктуры BioETL; для понимания сути достаточно знать, что они инкапсулируют параметры выполнения стадий, пути для файлов и результаты записи.)

Методы PipelineBase

```
extract(self, descriptor, options) -> pd.DataFrame
```

Extract – абстрактный метод извлечения данных, который должен быть реализован в каждом подклассе пайплайна ¹⁰. PipelineBase лишь определяет его сигнатуру, но не содержит реализации, так как способ получения данных зависит от конкретной сущности и источника.

- **Назначение:** Подключиться к внешнему источнику (API, база данных, файл) и загрузить сырьи данные для дальнейшей обработки. Метод возвращает **pandas DataFrame** с извлеченными записями ¹¹.
- **Параметры:** `descriptor` – объект дескриптора, определяющий что извлекать. Это может быть, например, идентификатор набора данных, параметры фильтрации или запрос. `options` – опции выполнения (например, настройки батч-процессинга, параметры логирования текущего запуска и т.п.).
- **Реализация в наследниках:** Каждая конкретная реализация пайплайна (например, ChEMBLActivityPipeline) определяет `extract` в соответствии с нужным API. Например, ChEMBL-пайплайны используют **ChEMBLExtractionService** и **дескриптор API** (с информацией об endpoint, фильтрах, страницах) для получения данных ¹². В PipelineBase через этот метод интегрируется любой источник: файл, HTTP API, SQL- запрос и т.д.
- **Ресурсы и клиенты:** Если для извлечения требуется создать внешний клиент (например, HTTP-сессию, API client), обычно это делается либо внутри `extract`, либо в методе-хуке `prepare_run`. PipelineBase предоставляет возможность зарегистрировать такие ресурсы (например, добавить объект клиента в `self._resources` или аналогичный механизм) и затем автоматически закрыть их после завершения пайплайна ¹³. Это гарантирует, что сетевые соединения или файлы не останутся открытыми. Например, ChEMBLBasePipeline в конструкторе сразу инициализирует **ChEMBLExtractionService** (включающий HTTP- клиент) и позже PipelineBase гарантирует его корректное закрытие ¹⁴ ¹³.

```
transform(self, df, options) -> pd.DataFrame
```

Transform – абстрактный метод преобразования данных, реализуемый в подклассах ¹⁵. На этапе трансформации сырьи данные очищаются, приводятся к нужной структуре и обогащаются необходимой информацией. Хотя PipelineBase не навязывает конкретную реализацию, общие подходы к трансформации унифицированы:

- **Нормализация и приведение типов:** В каждом пайплайне выполняется типизация столбцов, форматирование значений (например, даты в UTC, trimming строк) и заполнение дефолтных значений. Чтобы избежать дублирования этой логики в каждом классе, в проекте выделен базовый нормализатор. Например, для ChEMBL данные есть **BaseChEMBLNormalizer**, выполняющий типовое преобразование колонок по спецификации (ColumnNormalizationSpec): приведение типов, заполнение пустых значений по умолчанию, вычисление хешей строк (например, `hash_row`, `hash_business_key` для каждой записи) и пр. ¹⁶. Конкретные пайплайны могут использовать этот базовый нормализатор, расширяя его при необходимости специфичной обработкой (например, парсинг особых полей в Activity) ¹⁷.
- **Доменное обогащение (Enrichment):** PipelineBase предусматривает, что некоторые пайплайны добавляют дополнительные данные, не присутствующие напрямую в исходном источнике. Пример – добавление поля `chembl_release` (номер релиза базы ChEMBL) ко всем записям, или, для TestItem, добавление ID родительских веществ через внешний сервис (PubChem). Такой **enrichment** должен быть явно реализован в наследнике

при необходимости. В базовом классе (или промежуточном базовом вроде ChEMBLBasePipeline) обогащение оформлено как хук `domain_enrich`, который по умолчанию ничего не делает (noop)¹⁸. Это значит, что **обогащение отключено по умолчанию** и становится активным только если конкретный подкласс его переопределит. Такой подход делает обогащение опциональным – можно легко отключить эту стадию, не меняя основной логики пайплайна, просто не реализуя или не вызывая ее.

- **Единая структура трансформации:** Чтобы унифицировать описание преобразований, проект использует `descriptor` подход для схемы данных. Например, конфигурация пайплайна содержит раздел `fields` с описанием колонок (имя, тип, обязательность)¹⁹. Эти метаданные могут использоваться для автоматического построения Pandera-схемы и для настройки нормализации/преобразований. Таким образом, описав структуру единожды в конфиге, мы используем ее и для трансформации (на этапе нормализации/парсинга), и для валидации, и даже для генерации документации. PipelineBase (через SchemaProvider или ~~отдельные фабрики~~) приводит эти описания к единому интерфейсу, чтобы внутри метода `transform` программист оперировал не «магическими» названиями колонок, а, к примеру, объектом спецификации. В ChEMBL-пайплайнах применяется **ChEMBLDescriptorFactory** для получения описаний полей и стратегий парсинга^{20 21}, что помогает преобразовывать JSON от API в правильные колонки DataFrame (через маппинг полей)²².
- **Реализация в наследниках:** В конкретном пайплайне метод `transform` обычно выполняет следующие шаги:
- **Предобработка (опционально):** метод-хук `pre_transform` – может быть определен для каких-то общих действий до основной трансформации (по умолчанию пустой)¹⁸.
- **Доменное обогащение (опционально):** вызов `domain_enrich`, если требуются дополнительные данные (по умолчанию noop)¹⁸.
- **Основные преобразования:** приведение DataFrame к целевой схеме – применяются функции нормализации и маппинга. Например, вызов метода нормализации: `df = self.normalizer.normalize(df)` (где `normalizer` – экземпляр BaseChEMBLNormalizer или подобного класса). На этом шаге происходит приведение типов, вычисление новых технических полей (хеши, временные метки), переименование/переупорядочение колонок согласно схеме и другие трансформации, специфичные для сущности. Все пайплайны ориентированы на получение единообразной структуры данных, поэтому этот шаг стараются реализовать через общие компоненты.
- **Выравнивание структуры:** убедиться, что итоговые колонки и их порядок соответствуют ожидаемой схеме. Например, удалить лишние колонки, добавить отсутствующие с пустыми значениями, отсортировать колонки по заданному порядку. Это может быть частью нормализации или отдельная часть кода.

В результате метод `transform` возвращает преобразованный и очищенный DataFrame, готовый к проверке схемы¹⁵. Благодаря тому, что PipelineBase задаёт общие механизмы (хуки, нормализаторы, descriptors), разработчику конкретного пайплайна остается реализовать только уникальную бизнес-логику трансформации, не дублируя типовые операции²³.

Пример (pseudo-код):

```
class ChEMBLActivityPipeline(ChEMBLBasePipeline):  
    def transform(self, df: pd.DataFrame, options) -> pd.DataFrame:  
        df = self.pre_transform(df)           # общий пред-процессинг  
(наследуется, можно не трогать если не нужно)  
        df = self.domain_enrich(df)          # обогащение специфичными данными
```

```
(для Activity можно не переопределять, базовое добавит chembl_release)
df = self.normalizer.normalize(df) # нормализация базовых полей,
тиปизация, вычисление hash_row, etc.
# Дополнительная логика трансформации для Activity:
df["ligand_efficiency"] = df["le_field"].apply(parse_le) # пример
специфичной обработки поля
return df
```

(Здесь `Chemb1BasePipeline` уже реализует `pre_transform` и `domain_enrich` как хуки; `normalizer` – общий нормализатор ChEMBL. В `ActivityPipeline` мы дополняем только парсинг поля `ligand_efficiency`.)

`validate(self, df, options) -> pd.DataFrame`

Метод `Validate` выполняет проверку преобразованных данных на соответствие схеме и бизнес-правилам. В `PipelineBase` он реализован сразу (не абстрактный) – то есть общая логика валидации задается в базовом классе ²⁴.

- **Когда вызывается:** Стадия валидации запускается автоматически после трансформации, если для пайплайна определена схема и валидация не отключена (`PipelineBase` через план выполнения сам решает, включать ли этап `validate`) ²⁵.
- **Реализация:** `PipelineBase` использует компонент `validator` (реализация `ValidatorABC`, например `PanderaValidator`) и `schema_provider` (реализация `SchemaProviderABC`) для получения объекта схемы данных ⁶. Далее вызывается метод проверки, например `schema.validate(df)` (для Pandera) или соответствующий вызов Pydantic/JSON-схемы. Валидация обычно настроена в строгом режиме: проверяется наличие всех требуемых колонок, их типы, ограничения (диапазоны значений, форматы идентификаторов, уникальность ключа и т.п.) ²⁶. Также часто включена проверка порядка колонок, чтобы гарантировать детерминированность структуры ²⁷.
- **Результат:** Если `DataFrame` соответствует схеме, он возвращается без изменений ²⁸. В случае несоответствия бросается исключение (например, `SchemaValidationError`) с подробным отчетом о нарушениях ²⁹. Пайплайн может перехватить это исключение для логирования. Если `DataFrame` пуст, `validator` способен вернуть пустой `DataFrame` с правильными колонками (это предусмотрено, чтобы на выходе все равно получить корректно сформированный файл с заголовками) ³⁰.
- **Гибкость схемы:** Благодаря абстракции `SchemaProvider`, `PipelineBase` не привязан жестко к Pandera. Можно реализовать `schema_provider` для Pydantic-моделей или других схем, и `ValidatorABC` будет знать, как их применить. Таким образом, структура дескрипторов схемы унифицирована – `PipelineBase` просто запрашивает у провайдера "дай мне схему для этой сущности" и валидирует, не заботясь о деталях реализации ⁶. Это облегчает поддержку: например, базовые поля вроде `chembl_release`, `extracted_at`, `hash_row` могут быть определены в едином базовом классе схемы и унаследованы всеми сущностями ³¹ ³², что гарантирует единство метаданных во всех пайплайнах.

```
save_results(self, df, artifacts, options) -> WriteResult
```

Метод **SaveResults** отвечает за сохранение финального DataFrame и связанных артефактов на диск (либо в другое хранилище). Он реализован в базовом классе PipelineBase и не требует переопределения в большинстве случаев ⁹. Основные задачи этого метода:

- **Подготовка данных к записи:** Перед сохранением PipelineBase гарантирует детерминированность и целостность данных. В частности, DataFrame **сортируется** по определенным ключам (например, по первичному ключу или бизнес-ключу сущности) и/или по фиксированному порядку строк ³³. Это обеспечивает консистентность: каждый запуск пайплайна с одинаковым входом даст одинаково отсортированный выход, что важно для сравнения версий и вычисления хешей. Также проверяется и приводится порядок столбцов в соответствие со схемой, если это не было сделано ранее (обычно порядок уже проверен на этапе валидации при strict-mode) ²⁷.
- **Dry-run режим:** PipelineBase учитывает флаг `dry_run` (например, в `options`). Если включен `dry_run`, этап сохранения может быть пропущен или выполнен без фактической записи файлов. `build_stage_plan` формирует последовательность стадий таким образом, чтобы **не вызывать** `save_results` в **dry-run** ²⁵. Либо `save_results` внутри может проверить `options.dry_run` и вернуть имитацию результата, не сохраняя DataFrame. Это позволяет прогонять пайплайн полностью (включая валидацию) для проверки, но без воздействия на файловую систему ².
- **Атомарная запись:** Чтобы избежать частично записанных файлов или неконсистентного состояния, запись реализована атомарно. PipelineBase, как правило, делегирует запись `write_service`, например вызовом `self.write_service.write_dataset_atomic(df, artifacts)` ³⁴. Запись происходит сначала во временный файл, затем `os.replace` перемещает его на целевое место, заменяя старый файл. Таким образом, в случае сбоя во время записи, старые данные не затрагиваются, а в случае успеха новые данные заменяют старые мгновенно ³⁵. Этот механизм заложен в `UnifiedOutputWriter` и используется базовым классом ³⁶.
- **Артефакты и мета-информация:** Помимо основного датасета (обычно пишется в Parquet или CSV), PipelineBase обеспечивает создание дополнительной информации:
- **Файлы артефактов (artifacts):** Структура каталогов и имен файлов планируется заранее (обычно с помощью `WriteArtifacts`). Например, для ChEMBL Activity пайплайна создается папка с именем, содержащим идентификатор запуска (`run_id`), внутри которой файлы: `activity_<run_id>.parquet`, `meta.yaml`, отчеты качества данных и т.д. ³⁷. Эти пути формируются заранее (например, через `ArtifactPlanner`) и передаются в `save_results`.
- **Meta.yaml:** Файл метаданных, который сохраняется вместе с датасетом. В нем фиксируется информация о выполнении пайплайна: версия пайплайна или схемы, номер релиза источника (например, версия ChEMBL), время начала/окончания, количество записей, а также контрольные суммы (MD5, SHA256) файлов данных ³⁸. PipelineBase получает часть этих данных из конфигурации и хода выполнения, а часть вычисляется (например, хеш файла после записи).
- **Отчеты качества (QC reports):** Опционально, если включено в настройках, после основного сохранения могут генерироваться отчеты, например, о качестве данных или корреляции ³⁹. Это реализует либо сам `write_service`, либо дополнительный компонент, но PipelineBase отвечает за их вызов. Например, для ActivityPipeline генерируются `quality_report_activity_chembl.csv` и `correlation_report_activity_chembl.csv` ⁴⁰.
- **Возврат результата:** Метод возвращает объект `WriteResult` с информацией о том, что было записано (например, пути к файлам, количество записанных строк, размер файлов и

прочее) ⁹. Эта информация может использоваться внешними компонентами (например, CLI командой) для вывода пользователю или для дальнейшей обработки.

Таким образом, `save_results` инкапсулирует всю логику финальной стадии: от подготовки DataFrame до записи и логирования результатов. Обычно нет необходимости переопределять его в подклассах – все сущности пользуются единой реализацией записи, различаются лишь параметры (пути, имена файлов), задаваемые конфигурацией.

```
prepare_run(self, options) -> None и  
finalize_run(self, run_result) -> None
```

Это два **хук-метода**, определенные в `PipelineBase`, которые позволяют выполнять дополнительную логику до начала пайплайна и после его завершения. По умолчанию они не делают ничего (пустые методы) ⁴¹, и их переопределение опционально.

- `prepare_run` вызывается **перед началом стадии Extract** ⁴¹. В базовом классе это место зарезервировано для инициализации ресурсов и любых предварительных шагов. Например, если для пайплайна нужно подготовить окружение или удостовериться в наличии выходных директорий, это делается здесь. В контексте ChEMBL можно использовать `prepare_run` для вывода в лог информации о начале запуска, установки временной зоны (UTC) для дат, или для явного открытия соединения к API (в случае, если мы не хотим открывать его лениво при первом запросе).
- `finalize_run` вызывается **после завершения стадии Save (записи)** ⁴². Здесь удобно помещать очистку ресурсов: закрыть открытые соединения, клиенты, файлы, очистить временные файлы. `PipelineBase` в процессе выполнения пайплайна сам следит за ресурсами: после выполнения всех стадий **централизованно освобождает ресурсы**, вызывает у всех этапов метод `dispose()` (если этапы оформлены как отдельные объекты) ¹³. Однако `finalize_run` предоставляет дополнительную возможность наследнику выполнить кастомные действия завершения. Например, можно отправить уведомление о завершении загрузки, записать в базу факт обновления данных, или, как в случае ChEMBL, вызвать `extraction_service.dispose()` явно (если это не делается автоматически).
- В `finalize_run` обычно передается `run_result` – объект с результатами выполнения пайплайна (включает, например, сколько записей обработано, путь к файлам, были ли ошибки). На основе этой информации можно условно менять поведение завершения. По умолчанию `PipelineBase` игнорирует параметр.

Использование хуков не является обязательным, но они повышают расширяемость: вместо того, чтобы переписывать `run`-логику, можно аккуратно вставить дополнительные действия. Это соответствует принципу открытости/закрытости: базовый класс закрыт для модификации, но открыт для расширения через `override` хуков.

```
build_stage_plan(self, context, options) ->  
tuple[StageDescriptor, ...]
```

Метод формирования **плана стадий пайплайна**. `PipelineBase` определяет этот метод, чтобы собрать последовательность шагов, которые нужно выполнить при запуске пайплайна ²⁵. Он

анализирует текущие настройки и окружение и на их основе выстраивает план (обычно кортеж или список объектов-дескрипторов стадий):

- **Стандартный план:** По умолчанию возвращается кортеж стадий в порядке `extract` → `transform` → `validate` → `save` ²⁵, отражающий полный цикл ETL. Однако перед возвратом метод учитывает различные условия:
- Если **валидация** отключена или схема не задана, стадия `validate` может быть исключена из плана ²⁵.
- Если режим **dry_run** активирован, стадия `save` либо полностью исключается, либо заменяется на фиктивную (в плане может стоять заглушка, которая просто логирует результаты без записи). Таким образом, результат выполнения пайплайна не будет включать запись на диск при `dry_run` ²⁵.
- Возможны и другие условные этапы: например, если пайплайн поддерживает стадию обогащения как отдельный шаг (в некоторых реализациях `enrichment` мог бы быть отдельной Stage), PipelineBase мог бы включать или выключать ее в плане на основе конфигурации.
- **StageDescriptor:** Каждый элемент возвращаемого плана – объект, содержащий информацию о стадии: ссылку на метод (например, `PipelineBase.extract` или `PipelineBase.validate`), имя стадии, опции выполнения, и любые необходимые аргументы. Эти объекты затем используются механизмом исполнения пайплайна (Pipeline Runner) для последовательного вызова стадий. Например, `StageDescriptor` для `extract` будет содержать метод `self.extract` и аргумент `descriptor`, для `save` – метод `self.save_results` и аргументы `artifacts` и т.д. Обычно формирование этих дескрипторов – внутренняя деталь, вызывающаяся из общего запуска (`run()`), но важно, что `PipelineBase` централизует принятие решения какие этапы запускать.
- **Контекст (context):** параметр `context` может нести информацию о глобальном окружении запуска (например, профиле, общих сервисах, переменных окружения). PipelineBase может использовать его, чтобы пробросить нужные зависимости в стадии. Например, передать в `extract` готовый клиент, если он создан заранее, или предоставить в `save` путь для сохранения файлов из настроек окружения.
- **Расширяемость плана:** Хотя базовая реализация всегда `extract` → ... → `save`, наследник при необходимости может переопределить `build_stage_plan`, чтобы вставить дополнительные этапы. Например, если по архитектуре было решено вынести **enrichment как отдельную стадию**, можно переопределить этот метод: вернуть план `extract` → `transform` → `enrich` → `validate` → `save` в конкретном классе. Однако в текущей версии BioETL доменное обогащение реализуется обычно внутри `transform` (через `domain_enrich` хук), поэтому дополнительная стадия не требуется. Таким образом, разработчики получают единообразный конвейер по умолчанию, но имеют возможность подстроить его под нестандартные ситуации.

Расширение PipelineBase для конкретных сущностей

Благодаря общей архитектуре PipelineBase, создание новых пайплайнов сводится к реализации специфичных деталей, переопределяя минимально необходимые методы. Рассмотрим, как базовый класс расширяется на примере ChEMBL-пайплайнов, которые охватывают сущности Activity, Assay, Target, Document, TestItem:

- **ChEMBLBasePipeline:** Для семейства ChEMBL создается промежуточный базовый класс, наследующий PipelineBase ⁴³. Он инкапсулирует общую для всех ChEMBL-сущностей логику:

- Инициализация сервисов: при создании ChemblBasePipeline конфигурация проверяется на наличие обязательных параметров (например, `batch_size`, namespace кэша) ¹⁴. Затем создаются или подключаются общие сервисы: **ChemblExtractionService** (знает, как обращаться к API ChEMBL с учётом версии релиза) и **ChemblWriteService** (настройки сохранения результатов, например папка хранения для ChEMBL данных) ⁴⁴ ⁴⁵. Также инициализируется **ChemblDescriptorFactory** – фабрика, способная по конфигурам сущности выдать подходящий дескриптор для этапа извлечения ²⁰ (например, определить стратегию: вытаскивать все записи или по списку ID, и т.п.).
- Переопределение `extract`: В ChemblBasePipeline метод `extract` уже реализован универсально: он принимает дескриптор (описание, что вытаскивать) и делегирует работу **стратегии извлечения**. Внутри выбирается нужная стратегия через фабрику и вызывается `strategy.run()`, результатом чего будет DataFrame ¹². Таким образом, конкретным пайплайнам (Activity, Assay и др.) не нужно заново писать логику работы с API – они могут воспользоваться готовыми стратегиями, меняя лишь параметры.
- Реализация `transform`: ChemblBasePipeline определяет стандартный `transform` с использованием хуков ⁴⁶. Как описано выше, он последовательно вызывает `pre_transform`, затем `domain_enrich`, а далее – общие шаги нормализации и выравнивания. В самом ChemblBasePipeline после `domain_enrich` сразу может выполняться базовая нормализация (например, через BaseChembNormalizer) для приведения типов и добавления общих полей. Хуки `pre_transform` и `domain_enrich` объявлены здесь же: **по умолчанию они не делают ничего** ¹⁸, но их наличие позволяет конкретным пайплайнам вписаться в процесс:
 - Например, **ActivityPipeline** может не трогать `domain_enrich`, тогда по умолчанию добавится только релиз ChEMBL (возможно, `ChemblBasePipeline.domain_enrich` сам добавляет `chembl_release` через `extraction_service`). Зато Activity может внести особую обработку поля `ligand_efficiency` либо через переопределение `pre_transform` (чтобы распарсить поле до нормализации), либо после нормализации – в самом `transform` метода `ActivityPipeline`.
 - **TestItemPipeline** – особый случай: основной пайpline `TestItem` мог бы переопределять `domain_enrich`, чтобы выполнить обогащение данных о родительских веществах через внешний API (PubChem). В альтернативном "тонком" варианте (без обогащения) он этого не делает, оставляя `domain_enrich` пустым. Мы рассмотрим это на примере ниже.
- Таким образом, ChemblBasePipeline собирает все общие части: подключение к API, стандартная трансформация, общие схемы. Наследники же фокусируются только на различиях.
- **Конкретные пайплайны (ChemblActivityPipeline, ChemblAssayPipeline, ...)**: каждый наследует либо прямо ChemblBasePipeline, либо через еще какой-то промежуточный класс (например, могли бы быть специфичные базовые для похожих групп, но в нашем случае не указано). В них требуется:
 - Определить, при необходимости, собственный `transform`, если нужна дополнительная логика. Если базовой реализации достаточно (например, просто нормализовать базовые поля), можно даже не переопределять `transform`. Но чаще всего у каждой сущности есть особенности: разные наборы полей, форматы данных. Как минимум, может отличаться нормализатор или дополнительные вычисления. Поэтому, например, **ChemblTargetPipeline** может вызывать другой Normalizer или добавлять расчет дополнительных полей, но благодаря наследованию от ChemblBasePipeline, она получит `extract` и базовые части `transform` из родителя, переопределяя только то, что нужно.

- Настроить **схему валидации**: обычно реализуется вне класса – создается Pandera DataFrameModel для каждой сущности (например, `ActivitySchema`, `AssaySchema` и т.д.), которая наследует от базовой схемы и добавляет поля ⁴⁷ ₃₂. PipelineBase (через SchemaProvider) будет автоматически выбирать нужную схему по имени сущности (например, по `entity_name` в конфиге). Таким образом, конкретный класс пайплайна не кодирует схему напрямую – она приходит из реестра схем, обеспечивая единообразие.
- Регистрация пайплайна**: новый класс должен быть зарегистрирован в фабрике или CLI, чтобы система могла его запустить по имени. Это делается вне самого класса (например, через декоратор или путем добавления в mapping). В контексте вопроса достаточно знать, что классы названы по шаблону `<Entity><Provider>Pipeline` и соотносятся с конфигами (например, `activity_chembl`).
- Enrichment настройка**: Если для сущности предусмотрено обогащение, разработчик решает, включать его или нет. Например, `TestItemPipeline` может иметь два варианта: полный (с обогащением PubChem) и упрощенный. Они могут быть реализованы двумя разными классами или параметризоваться. За счет того, что `domain_enrich` – хук, можно реализовать **включаемое обогащение**:
 - Полный вариант переопределяет `domain_enrich` и внутри вызывает, к примеру, `self.pubchem_client.enrich_parent_ids(df)` – функцию, которая обращается к PubChem для поиска parentId для каждого тест-айтема. Результат – DataFrame с добавленной колонкой `parent_id` и объект статистики обогащения (как описано в документации) ⁴⁸.
 - «Тонкий» вариант вообще не переопределяет `domain_enrich`, либо явно отключает обогащение через флаг конфигурации. Тогда используется только базовая трансформация ChEMBL (получаются только поля, пришедшие из ChEMBL, плюс базовые из `chembl_release`). Такой подход действительно реализован: существует `Chemb1TestItemThinPipeline`, который не требует PubChem клиента, работает быстрее и выдаёт минимальный набор данных ⁴⁹ ₅₀.

Итого, расширение PipelineBase сводится к тому, что **общие части реализованы один раз** (в PipelineBase и, при необходимости, в промежуточных базовых классах), а в конкретных классах разработчик пишет только специфическую логику стадии извлечения (если нельзя все покрыть общей стратегией) и стадии трансформации (для особенностей данных). Стадии валидации и сохранения, а также работа с метаданными, логированием, управлением ресурсами – едины для всех и повторно используются из PipelineBase. Такой дизайн следуют принципу DRY и обеспечивает согласованность поведения пайплайнов ¹.

Пример: минимальный шаблон `Chemb1TestItemPipeline` без обогащения

Чтобы проиллюстрировать использование интерфейса PipelineBase, рассмотрим упрощенную реализацию пайплайна для сущности **TestItem (ChEMBL)**, где отключено внешнее обогащение. Предположим, что у нас есть базовый класс `Chemb1BasePipeline` (описанный выше) с готовой логикой. Тогда определение нового пайплайна может выглядеть так:

```
from bioetl.pipelines.chembl.common.base import Chemb1BasePipeline

class Chemb1TestItemPipeline(Chemb1BasePipeline):
    """ETL-пайплайн для ChEMBL TestItem без внешнего обогащения (PubChem)."""

```

```

def extract(self, descriptor, options):
    # Используем ChemblExtractionService через базовый класс:
    # ChemblBasePipeline.setup уже создал extraction_service на основе
config.
    # descriptor обычно None, т.к. извлекаем все данные по endpoint.
    df = self.extraction_service.fetch_all("test_item") # псевдо-код
метода
    self.logger.info(f"Extracted {len(df)} test items")
    return df

    # Метод transform не переопределяем, используем реализацию из
ChemblBasePipeline:
    # Он сам вызовет pre_transform (noop) и domain_enrich (noop, т.к. мы не
переопределили).
    # Затем выполнит базовую нормализацию (ChemblBasePipeline/нормализатор
приведет типы, добавит chembl_release, hash_row и т.д.)
    # Если бы требовалась дополнительная обработка, можно было бы
переопределить или воспользоваться хуком.

# Validate также не переопределяем – базовый класс применит Pandera-схему
TestItem из реестра.

# Save_results не переопределяем – сохранение, сортировка, метаинформация
выполняются общим механизмом.

```

В этом шаблоне мы фактически воспользовались большинством возможностей базового класса:

- `extract` делегирован к сервису ChEMBL (который знает, как получить TestItem данные через API). Заметим, мы **не заботимся о деталях HTTP** – это инкапсулировано в `extraction_service`, настроенном в `ChemblBasePipeline`¹⁴. Достаточно вызвать метод, передав, например, имя сущности или дескриптор.
- Мы **не переопределяем** `transform` и `domain_enrich`, значит никаких дополнительных данных из PubChem не добавляем – пайpline ограничится базовыми полями ChEMBL. Благодаря тому, что `domain_enrich` по умолчанию пустой, никакого лишнего действия не произойдет¹⁸. Базовый нормализатор добавит стандартные колонки (`chembl_release`, `hash_row`, `timestamps`) и приведет типы.
- Валидация и сохранение полностью выполняются родительским классом. Валидация проверит DataFrame TestItem по заранее описанной схеме (например, убедится, что присутствуют нужные поля, такие как `test_item_id`, `chembl_release`, и что они правильного типа). Сохранение запишет результат в Parquet/CSV, создаст `meta.yaml` и другие артефакты, отсортировав данные по `test_item_id` или другому ключу для детерминированности. Поскольку мы не указали `dry_run`, данные реально сохранятся, иначе `build_stage_plan` просто бы исключил этот шаг²⁵.

Выход: `ChemblTestItemPipeline` в «тонкой» конфигурации (без обогащения) получился очень лаконичным – мы реализовали только метод `extract` (и то минимально), полностью полагаясь на `PipelineBase` и `ChemblBasePipeline` в остальном. Согласно документации, такой тонкий пайpline **работает быстрее и требует меньше зависимостей**, так как не обращается к внешним сервисам вроде PubChem⁵⁰. В случаях, когда дополнительное обогащение всё же

нужно, можно расширить этот же класс или создать альтернативный, переопределив `domain_enrich` для интеграции с PubChem (например, используя `ParentEnrichmentPreparation / Result` для добавления `parent_id`, см. документацию) ⁴⁸.

Таким образом, архитектура PipelineBase обеспечивает единообразный процесс ETL для разных сущностей, позволяя гибко отключать или добавлять этапы (например, обогащение) по необходимости, не дублируя код. Все пайплайны ориентированы на работу с DataFrame и следуют одному интерфейсу, что упрощает поддержку и расширение системы. Это соответствует целям проекта BioETL по снижению дублирования и упорядочиванию ETL-процессов ⁵¹ ¹.

Источники:

- Описание базового класса PipelineBase и его функций ² ¹³
 - Архитектура и паттерны переиспользования в BioETL ¹ ¹⁶
 - Детальный flow ChEMBL Activity Pipeline (ETL этапы) ⁵² ⁵³ ²⁶ ⁷
 - Базовый класс ChEMBLBasePipeline и его хуки (`pre_transform`, `domain_enrich`) ¹⁸ ⁴⁶
 - Пример тонкого пайплайна без обогащения (TestItem Thin) ⁵⁰ ⁴⁹
 - Механизм схем валидации (SchemaProvider, Pandera DataFrameModel) ⁶ ⁴⁷
 - Документация по atomic write и метаинформации ⁷ ³⁸
-

¹ ⁴ ¹⁶ ¹⁷ ²³ ³¹ ³² ⁴⁷ ⁵¹ 04-architecture-and-duplication-reduction.md

<https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/project/04-architecture-and-duplication-reduction.md>

² ⁹ ¹⁰ ¹¹ ¹³ ¹⁵ ²⁴ ²⁵ ³⁶ ⁴¹ ⁴² 00-pipeline-base.md

<https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/02-pipelines/00-pipeline-base.md>

³ ¹⁹ adding-new-pipeline.md

<https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/guides/adding-new-pipeline.md>

⁵ ¹² ¹⁴ ¹⁸ ²⁰ ²¹ ⁴³ ⁴⁴ ⁴⁵ ⁴⁶ 05-chembl-base-pipeline.md

<https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/02-pipelines/chembl/common/05-chembl-base-pipeline.md>

⁶ 20-schema-provider-abc.md

<https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/reference/abc/20-schema-provider-abc.md>

⁷ ⁸ ²² ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³³ ³⁴ ³⁵ ³⁷ ³⁸ ³⁹ ⁴⁰ ⁵² ⁵³ data-flow.md

<https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/architecture/data-flow.md>

⁴⁸ 05-testitem-chembl-parent-enrichment-result.md

<https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/02-pipelines/chembl/testitem/05-testitem-chembl-parent-enrichment-result.md>

⁴⁹ ⁵⁰ 02-testitem-chembl-thin-pipeline.md

<https://github.com/SatoryKono/BioactivityDataAcquisition/blob/47444b81a28f5b8397ab197f5bc608866d84a7d5/docs/02-pipelines/chembl/testitem/02-testitem-chembl-thin-pipeline.md>