ChatGPT

Резюме основных проблем и предложений

Документация содержит несколько несоответствий и дублирований, усложняющих поддержку. В частности, выявлены нарушения собственных правил именования (некорректные имена файлов), противоречия между гайдами и фактической структурой кода (например, ссылки на несуществующие файлы), а также дублирование содержания в разных документах. Терминология используется непоследовательно (одно и то же понятие называется по-разному в разных местах). В архитектуре проекта, по описаниям, наблюдаются повторяющиеся решения для разных провайдеров данных, что приводит к дублированию кода и конфигураций. Ниже перечислены конкретные проблемы, предложения по их исправлению и рекомендации по улучшению архитектуры с целью уменьшения повторяемости.

Найденные ошибки и несоответствия документации

- **Нарушение политики именования файлов документации.** Некоторые файлы в `docs/` не соответствуют заявленным правилам именования: вместо **lower-kebab-case** с префиксом используются заглавные буквы и подчеркивания. Например, файлы `MEMORIES.md` и `RULES_QUICK_REFERENCE.md` не соблюдают требование “все имена файлов в lowercase, слова через дефис, никаких underscore” [1] [2] . Это прямое нарушение зафиксированного стайлгайда.

- **Дублирование справочных материалов.** В папке styleguide присутствуют сразу несколько кратких сводок правил, существенно пересекающихся по содержанию: **00-rules-summary.md**, **RULES_QUICK_REFERENCE.md** и **MEMORIES.md**. Все они перечисляют одни и те же правила (форматы имен, суффиксы классов, префиксы функций и т.д.), часто буквально повторяя пункты друг друга [3] [4] . Поддержание нескольких копий одной информации приводит к рассинхронизации – правки в правила приходится дублировать, есть риск, что файлы разъедутся со временем.

- **Несоответствие описания структуры кода реальной организации.** Политика по ABC/Default/Impl предполагает определённое расположение файлов, которое не отражает текущее состояние проекта. Например, в документе **01-new-entity-implementation-policy.md** указано, что общие контракты находятся в `base/contracts.py`, а дефолтные фабрики – в файлах `factories.py` каждого домена (провайдера) [5] . В реальном же коде, судя по обзору клиентского слоя, таких файлов может не быть: вместо них используется единый модуль `factory.py` и единый реестр клиентов, а в папках конкретных источников (ChEMBL, PubChem и т.д.) присутствуют только файлы клиентов (`client.py`) [6] . То есть документация ссылается на `contracts.py` и `factories.py` по каждому домену, тогда как фактические файлы названы иначе (например, `client_abc.py` вместо `contracts.py`) или совмещены. Это создает путаницу: правила не совпадают с реальной структурой проекта.

- **Несогласованные внутренние ссылки и нумерация документов.** В некоторых местах документы ссылаются друг на друга с использованием устаревших путей или номеров. Так, **02-new-entity-naming-policy.md** (Полная политика именования) в разделе “Sources” указывает на файлы по пути `docs/styleguide/...` с номером “11-naming-policy.md” [7] ,

хотя в репозитории эти файлы находятся в `docs/00-styleguide/...` и имеют номера “02-...” или “10-...”. Аналогично, в README приведены актуальные пути (`docs/00-styleguide/02-new-entity-naming-policy.md` и др.) [8], которые не совпадают с упоминаниями внутри некоторых документов. Это противоречие может привести к битым ссылкам и затрудняет навигацию по документации.

- **Непоследовательная терминология.** Разные документы используют различные термины для одних и тех же сущностей. К примеру, внешние источники данных называются то “домен” (в контексте клиентов) [9], то “провайдер” (в контексте пайплайнов) [10]. Еще пример: абстрактные классы могут именоваться “ABC”, “Protocol” или “контракт” в разных разделах. Хотя по смыслу это одно и то же, отсутствие единообразия может сбивать с толку читателя. Термин “Pipeline” в некоторых случаях обозначает и весь ETL-процесс, и конкретный класс/модуль `run.py`, что тоже нужно оговаривать четче.

- **Несогласованность в описании исключений из правил.** В разных местах политики по именованию по-разному указано, как фиксировать исключения. Одна часть требует регистрировать все отклонения от MUST-правил в YAML-файле `configs/naming_exceptions.yaml` [11], другая подразумевает документирование исключений в специальном Markdown (`docs/exceptions.md`) с обоснованием и сроком действия [12]. Эти требования дублируют друг друга и могут конфликтовать. На данный момент файл **docs/exceptions.md** не обнаружен в репозитории, то есть правила об обработке исключений прописаны, но место их ведения не определено однозначно.

- **Отсутствие явного обзора архитектуры (ADR/C4).** В предоставленных материалах нет отдельного документа, описывающего общую архитектуру системы на высоком уровне (например, диаграммы C4: контекст, контейнеры, компоненты) или фиксирующего архитектурные решения (ADR). Есть точечные диаграммы классов для модуля clients [13] и описание базового устройства пайплайнов [14], но целостного обзора (какие основные компоненты системы и как они взаимодействуют) не представлено. Это затрудняет понимание общей картины и мотивов решений. Например, введение **UnifiedAPIClient** или **SchemaRegistry** описано в отдельных документах, но нет единого ADR, объясняющего, зачем они были нужны архитектурно.

Предложенные исправления в документации

- **Приведение имен файлов в соответствие с политикой.** Переименовать вспомогательные файлы styleguide в lowercase-kebab-case. Например, `MEMORIES.md` можно переименовать в `00-memories.md` или `memories.md`, а `RULES_QUICK_REFERENCE.md` – в `rules-quick-reference.md` (либо интегрировать его содержание в `00-rules-summary.md`). Это устранит нарушения правил именования [1] и сделает имена файлов однородными. В самом **00-naming-conventions.md** стоит добавить явное указание, что *любые* файлы документации (не только pipeline docs) должны соблюдать эти же требования, без исключений.

- **Объединение дублирующих справочников.** Устранить параллельное существование нескольких сводок правил. Рекомендуется выбрать один источник правды для краткого перечня правил – например, оставить только **RULES_QUICK_REFERENCE.md** (или, наоборот, **00-rules-summary.md**) – а остальные удалить или превратить в сгенерированные вложения. Можно, к примеру, включить раздел “Summary” прямо в README или основной styleguide, вместо поддержки отдельного файла **MEMORIES.md**. В результате все ключевые

правила будут в одном месте, что упростит обновление. Если требуется разные форматы (одностраничная шпаргалка и развёрнутый документ), можно генерировать шпаргалку автоматически из полного документа, помечая соответствующие секции тегом `<!-- generated -->` [15]. Главное – исключить ручное дублирование содержания, чтобы правки правил вносились один раз.

- **Актуализация структуры в гайдах под фактический код.** Исправить документы styleguide, описывающие архитектуру "ABC/Default/Impl", в соответствии с реальной организацией модулей. В **01-new-entity-implementation-policy.md** нужно заменить ссылки на несуществующие файлы. Например, вместо `src/bioetl/clients/base/contracts.py` указать актуальный `src/bioetl/clients/base/client_abc.py` (если в нем сосредоточены базовые Protocol) [9] [16]. Аналогично, уточнить, где именно находятся Default-фабрики: если проект использует централизованный `src/bioetl/clients/factory.py` и **ClientRegistry** для создания клиентов, то в политике следует прямо это сказать, убрав упоминание вымышленных файлов `clients/<domain>/factories.py`. Это устранит расхождения между документацией и кодом, сделав руководство применимым на практике. Если же планируется привести код в соответствие с изначальной политикой (распределить фабрики по подпапкам), то такое решение тоже должно быть отражено: например, в виде пометки "*(планируется рефакторинг расположения фабрик по провайдерам)*".

- **Корректировка внутренних ссылок и номеров документов.** Необходимо проверить и исправить все перекрестные ссылки внутри документации. В частности, в **02-new-entity-naming-policy.md** (политика именования) ссылки на источники стоит изменить с `docs/styleguide/..` на актуальный путь `docs/00-styleguide/..`, либо вообще убрать указание промежуточной папки, если структура может меняться. Номер документа "11-naming-policy.md" следует заменить на фактический `02-new-entity-naming-policy.md` [7], чтобы не сбивать читателя. В целом, нумерация в названиях файлов и упоминаниях должна быть синхронизирована: например, если документ **10-documentation-standards.md** называется так, то и ссылки на него в тексте должны использовать "10", а не другой индекс. Рекомендуется завести единый `INDEX.md` для styleguide-раздела, где перечислить все правила с правильными названиями – это снизит вероятность ошибки при ручной правке ссылок.

- **Унификация терминологии.** Во всех документах следует привести термины к одному словарю. Для обозначения внешнего источника данных предпочтительнее использовать один термин – например, везде **"провайдер"** (как уже принято в документации по пайплайнам) вместо разнородных "домен/провайдер/источник". В тексте **01-new-entity-implementation-policy.md** при описании структуры клиентов можно заменить слово "domain" на "provider" [9], чтобы совпадало с остальными разделами. Также стоит дать краткий глоссарий: указать, что **ABC (Abstract Base Class)** = контракт = протокол – и далее по тексту придерживаться одной аббревиатуры (например, "ABC") для консистентности. Все абстракции должны именоваться одинаково: если введено понятие "ABC/Protocol", то использовать оба термина вместе каждый раз необязательно – достаточно одного, но последовательно. Эти правки сделают требования однозначными: читатель не будет гадать, тождественны ли "провайдер" и "домен" или чем протокол отличается от ABC.

- **Единый подход к фиксации исключений из правил.** Следует выбрать один канал регистрации исключений и отразить это во всех документах. Если предполагается технический YAML (`configs/naming_exceptions.yaml`) как основной список

игнорируемых нарушений, то **01-new-entity-implementation-policy.md** и другие документы должны ссылаться на него и не упоминать `docs/exceptions.md`. В противном случае – если важна человеческо-читаемая история исключений – нужно добавить в репозиторий файл **docs/exceptions.md** и в CI/политику направлять разработчиков заполнять его при необходимости. Можно реализовать оба механизма (YAML для CI и Markdown для обзора), но тогда документацию надо явно уточнить: например, "*исключения MUST быть добавлены в* `configs/naming_exceptions.yaml`*, а причины и сроки – описаны в* `docs/exceptions.md`". В настоящее время такая связка не прояснена. Исправление этого несоответствия сделает процесс обхода правил прозрачным и понятным участникам проекта [12] [11].

- **Добавление обзора архитектуры и решений.** Рекомендуется подготовить отдельный раздел или документ с описанием общей архитектуры системы. Это может быть страница "Architecture Overview" с C4-диаграммой контекста и контейнеров, где показано, как взаимодействуют **core**, **pipelines**, **clients**, **QC** и прочие слои. Кроме того, имеет смысл завести **ADR (Architecture Decision Records)** для ключевых решений. Например, оформить ADR для введения единого HTTP-клиента или для выбора Pandera в качестве механизма валидации. Сейчас подобные решения упомянуты фрагментарно (в README или узких документах), но не связаны в единый нарратив. Наличие таких документов устранит неопределенность: новые участники команды смогут быстро понять, **почему** архитектура организована именно так. Кроме того, фиксируя решения, легче отследить противоречия – чтобы одно улучшение (например, унификация клиента) не было случайно нивелировано другим разработчиком из-за отсутствия информации. Подготовка архитектурного обзора сделает документацию самодостаточной и удобной для сопровождения.

Рекомендации по архитектуре и снижению дублирования

- **Выделение и рефакторинг общих компонентов.** Текущая архитектура уже содержит задел для устранения дублирования: введены **PipelineBase** для базовой логики выполнения пайплайна, **UnifiedAPIClient** для унификации работы с внешними REST API, **UnifiedOutputWriter** для записи результатов и т.д. Необходимо развивать эту идею и убеждаться, что повторяющийся код действительно перемещается в общие модули. Например, если в нескольких пайплайнах реализованы похожие этапы преобразования или схожая обработка ошибок, стоит вынести их в модуль **core** или в базовые классы и абстракции. Так, для повторяющихся операций трансформации данных можно обеспечить реализацию интерфейса **TransformerABC** и предоставить набор типовых **Transformer**-классов, которые могут переиспользоваться разными пайплайнами (вместо копирования одних и тех же функций). Аналогично, для валидации данных можно иметь единый класс-валидатор, использующий Pandera-схему, и вызывать его во всех пайплайнах, где нужна стандартная проверка схемы, вместо написания отдельной логики в каждом пайплайне. В результате пайплайны будут различаться только бизнес-логикой, а шаблонный код (загрузка конфигурации, цикл ETL, логгирование, обработка ошибок) будет централизован, что соответствует целям политики (DRY, детерминизм и пр.) [17] [14].

- **Унификация конфигураций пайплайнов.** Обратить внимание на возможное дублирование в YAML-конфигах. Сейчас для каждого пайплайна заведён свой файл конфигурации (например, `configs/pipelines/chembl/activity.yaml`), в котором могут повторяться одни и те же структуры: секции хранения данных, типовые параметры клиента (таймауты, retries) и т.д. [18] [19]. Уже реализована поддержка *профилей*

(development, production и пр.) и *overrides* для override-значений через CLI [20] [21]. Чтобы ещё уменьшить дублирование, можно вынести общие для всех (или группы) пайплайнов настройки в отдельные конфиги. Например, базовые пути `storage_path`, `cache_path` и прочие – в профиль по умолчанию или глобальный файл, а в индивидуальных YAML не дублировать их. Также можно воспользоваться тем, что каждому провайдеру соответствует клиентский YAML (например, `clients/config/<provider>.yml` с описанием API endpoints) – убедиться, что пайплайновый конфиг не повторяет информацию, которая уже есть в YAML клиента. Возможно, стоит связать их явнее: например, в конфигурации пайплайна хранить только ссылку на ключ в клиентском конфиге (имя ресурса), вместо копирования URL или параметров запроса. Это обеспечит **единственное место** для изменения при обновлении API провайдера. В идеале, параметры, общие для всех сущностей провайдера, должны быть определены на уровне провайдера, а специфичные – на уровне пайплайна. Таким образом удастся унифицировать пайплайны и их конфиги: уменьшить объем настроек, которые нужно править при добавлении нового пайплайна того же типа.

- **Четкое разграничение ответственности между слоями.** Продолжая предыдущий пункт, важно, чтобы каждый слой системы занимался “своим” делом, и функциональность не дублировалась на нескольких уровнях. Судя по документации, в проекте уже намечена такая структура: слой **clients** отвечает за получение данных от внешних сервисов, **pipelines** – за последовательность ETL-шагов и бизнес-логику обработки, **core** – за общие утилиты (логирование, конфигурирование, схемы и пр.), **qc** – за контроль качества, **tools** – за вспомогательные скрипты и т.д. Следует проверить, что нигде не происходит размытия границ. Например, чтобы код работы с HTTP/API не размазан по разным пайплайнам, а полностью инкапсулирован в клиентах (UnifiedAPIClient + специализированные RequestBuilder/ResponseParser для конкретного API) [22] [23]. В пайплайнах же не должно оставаться низкоуровневого кода запросов – только вызовы методов клиентов. Если где-то пайплайн напрямую использует `requests` или собственную логику пагинации, это сигнал к рефакторингу: надо использовать существующие адаптеры и ABC. Другой пример – **валидация и QC**: убедиться, что проверки данных (Pandera схемы, DQ rules) выполняются либо в явном этапе `validate()` пайплайна, либо через общую инфраструктуру (SchemaRegistry, QualityReport), но не смешаны с этапом трансформации. Документация упоминает SchemaRegistry [24] и DQ-правила, что подразумевает централизованный подход. Нужно ему следовать, чтобы каждый новый пайплайн не изобретал свой способ валидировать данные. В итоге такое упорядочение ответственности между слоями устранит скрытое дублирование, когда схожий функционал реализован дважды в разных частях системы. Вместо этого один слой (и, желательно, один модуль) будет содержать реализацию, а остальные слои просто пользоваться им.

- **Унификация схематизации и данных между пайплайнами.** Документы указывают, что для разных сущностей (activity, assay, target и т.д.) и разных провайдеров (ChEMBL, PubChem...) определены свои Pandera-схемы данных [25] [26]. Имеет смысл оценить, нет ли среди них существенного пересечения, которое можно вынести в общие компоненты. Если, например, несколько провайдеров предоставляют данные об одних и тех же сущностях (скажем, “Document” из PubMed и из CrossRef), стоит рассмотреть введение единой **базовой схемы** для “Document” и специфических подтипов для каждого провайдера, наследующих эту базу. Это обеспечит одинаковую структуру выходных данных и облегчит последующую консолидацию данных из разных источников. Кроме того, единые схемы позволят не дублировать описание общих колонок (ID, название, даты и пр.) в нескольких файлах – вместо этого эти поля будут определены в одном месте. По схожему принципу можно унифицировать и обработку: например, если нормализация

каких-то полей (та же дата или идентификаторы) одинаковая для всех источников, можно вынести её в **core** (утилиты или BaseNormalizer), чтобы не писать один и тот же код в **ActivityNormalizer**, **AssayNormalizer** и т.п. Обзор документов по ChEMBL показывает, что для этого провайдера выделены общие компоненты (например, `chembl-common-pipeline.md`, `chembl-extraction-service.md`) [27]. Аналогично, можно оценить вынос общих частей на уровень *нескольких провайдеров*. Если какие-то сущности универсальны (например, все биоактивности имеют схожие атрибуты независимо от источника), их обработка должна быть тоже максимально универсальной. Это повысит **повторное использование кода** и снизит риск, что в одном пайплайне что-то улучшили (например, алгоритм дедупликации), а в другом забыли, получив рассинхрон.

- **Повторное использование уже реализованных абстракций вместо создания новых.** В проекте задан целый каталог ABC (контрактов) с разнообразными ролями – CacheABC, RetryPolicyABC, PaginatorABC, WriterABC и т.д. [28] [29]. Это большой плюс, так как новые компоненты можно строить на основе существующих контрактов. Рекомендуется при разработке новых функциональностей сперва искать, нет ли уже подходящей абстракции, прежде чем добавлять новую. Документация по UnifiedAPIClient, например, показывает, что он опирается на несколько контрактов (RequestBuilderABC, PaginatorABC, ResponseParserABC) [22] [30] – и это правильный подход. Надо придерживаться его повсеместно: если для задачи контроля ошибок уже есть **ErrorPolicyABC**, не заводить отдельный механизм обработки ошибок в конкретном пайплайне, а имплементировать существующий интерфейс (или расширить его, если не хватает возможностей). Каждый новый пайплайн или клиент должен стараться использовать Default/Impl из **abc_impls.yaml** там, где это возможно, вместо написания “с нуля”. Такой принцип **реюза** сократит дублирование логики: одна реализация, зарегистрированная в реестре, может обслуживать множество компонентов. А если потребуется модификация в будущем (например, обновить алгоритм RetryPolicy), ее достаточно будет внести в одном месте, что сразу улучшит все части системы, где она используется.

- **Организация кода без избыточных слоев.** Наконец, при развитии архитектуры важно соблюдать баланс: не вводить новых сущностей, если задачу можно решить перестройкой существующих. Документация требует не предлагать новых слоев без необходимости – этот же подход стоит применять при рефакторинге. Например, прежде чем выносить какую-то часть логики в отдельный сервис/слой, следует проверить: нельзя ли разместить её в уже имеющемся модуле **core** или оформить как класс в рамках текущего слоя. Допустим, возникает потребность в дополнительном этапе обработки данных – возможно, это просто еще один **Stage** в существующем Pipeline, а не новый слой. Или, скажем, необходимо собирать дополнительные метрики – вместо создания отдельного “MetricsCollector” слоя, можно расширить **QC** или использовать **PipelineHookABC** для этой цели [31]. Такой подход обеспечит, что архитектура останется понятной и плоской, а код – более сконцентрированным. Новые сущности действительно нужны лишь тогда, когда несколько компонентов начинают повторять одни и те же функции и их сложно встроить в текущие классы. Если же такая ситуация наступает (например, обнаруживается дублирование конфигурационного кода и его уже нельзя удачно вписать в существующий **ConfigResolver**), тогда обоснованное введение новой сущности должно быть сразу задокументировано: что она решает и почему рефакторинг старой неудовлетворителен. Это возвращает нас к идее ADR – фиксировать решения и их обоснование, чтобы в будущем не появлялись “лишние” слои по недоразумению.

В целом, выполненные правки в документации устранят фактические ошибки и противоречия, сделают требования однозначными, а удаление дублирующего контента упростит

сопровождение правил. Рекомендуемые архитектурные улучшения нацелены на то, чтобы проект **BioETL** оставался единым целым: общие механизмы – централизованными, повторяющиеся шаблоны – унифицированными, а каждый новый компонент встроен в существующую структуру, а не дублирует её. Это позволит уменьшить объем кода и конфигураций, которые нужно поддерживать, и повысит согласованность системы в долгосрочной перспективе.

---

[1] [10] 00-naming-conventions.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/20768daae3f909b6f02b071b137ca4eead3abb30/docs/00-styleguide/00-naming-conventions.md

[2] [8] [25] [26] [27] README.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/20768daae3f909b6f02b071b137ca4eead3abb30/README.md

[3] [11] MEMORIES.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/20768daae3f909b6f02b071b137ca4eead3abb30/docs/00-styleguide/MEMORIES.md

[4] [15] 00-rules-summary.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/20768daae3f909b6f02b071b137ca4eead3abb30/docs/00-styleguide/00-rules-summary.md

[5] [9] [12] 01-new-entity-implementation-policy.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/20768daae3f909b6f02b071b137ca4eead3abb30/docs/00-styleguide/01-new-entity-implementation-policy.md

[6] [16] 00-clients-overview.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/20768daae3f909b6f02b071b137ca4eead3abb30/docs/clients/00-clients-overview.md

[7] 18-new-entity-naming-policy.md
https://github.com/SatoryKono/bioactivity_data_acquisition/blob/98f17f432532cddd56d0a07fd4796b37c54677ec/docs/18-new-entity-naming-policy.md

[13] 19-clients-diagrams.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/20768daae3f909b6f02b071b137ca4eead3abb30/docs/clients/19-clients-diagrams.md

[14] [24] 00-activity-chembl-overview.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/20768daae3f909b6f02b071b137ca4eead3abb30/docs/02-pipelines/chembl/activity/00-activity-chembl-overview.md

[17] 10-documentation-standards.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/20768daae3f909b6f02b071b137ca4eead3abb30/docs/00-styleguide/10-documentation-standards.md

[18] [19] [20] [21] 02-logging-and-configuration.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/20768daae3f909b6f02b071b137ca4eead3abb30/docs/02-pipelines/02-logging-and-configuration.md

[22] [23] [30] 03-unified-api-client.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/d76b63a88157ada22fa448587fbba3cf639f3a02/docs/02-pipelines/03-unified-api-client.md

[28] [29] [31] INDEX.md
https://github.com/SatoryKono/BioactivityDataAcquisition/blob/d76b63a88157ada22fa448587fbba3cf639f3a02/docs/01-ABC/INDEX.md