NovaData LMS

Документация проекта

Область документа: `lms-novadata` , `SQLChecker` , `code_checker`
Назначение: зафиксировать текущую архитектуру проекта, основные сценарии работы, интеграции с сервисами проверки и известные ограничения.

Документ подготовлен как версия технической документации. Он не описывает каждый класс и функцию, а фиксирует ключевые части системы, без которых невозможно сопровождать, разворачивать и развивать проект.

Связанные документы:

- `README_DOCUMENTATION.md` - индекс комплекта документации и рекомендуемый порядок чтения.
- `DEPLOYMENT_GUIDE.md` - подробная инструкция по развертыванию.
- `ENVIRONMENT_VARIABLES.md` - актуальная схема переменных окружения.
- `API_CONTRACTS.md` - API-контракты LMS/checkers.
- `OPERATIONS_RUNBOOK.md` - регламент эксплуатации и восстановления после сбоев.
- `PROGRAMMING_TASK_AUTHOR_GUIDE.md` - руководство по созданию programming-заданий.
- `SQL_TASK_AUTHOR_GUIDE.md` - руководство по созданию SQL-заданий.

Оглавление

1. Назначение проекта

NovaData LMS - учебная платформа для создания курсов, прохождения уроков и автоматической проверки учебных заданий. Проект состоит из трех связанных частей:

- `lms-novadata` - основное Django-приложение LMS.
- `SQLChecker` - внешний сервис проверки SQL-решений.
- `code_checker` - внешний сервис проверки программного кода в изолированных Docker-окружениях.

Основная задача системы - предоставить авторам курсов инструменты для создания учебного контента, а студентам - интерфейс для прохождения материалов и отправки решений на автоматическую проверку.

2. Состав проекта

2.1. lms-novadata

Основное веб-приложение на Django. Отвечает за:

- авторизацию пользователей;
- управление курсами, модулями, уроками и шагами;
- интерфейсы студента, автора и администратора;
- хранение прогресса прохождения;
- создание попыток проверки SQL- и programming-заданий;
- интеграцию с внешними сервисами проверки;
- хранение очереди проверок и истории результатов.

Ключевые каталоги:

- `lms_project/apps/core` - основная логика курсов, уроков, прогресса, отображения шагов.
- `lms_project/apps/author` - интерфейс автора курса.
- `lms_project/apps/adm` - административные интерфейсы.

- `lms_project/apps/api` - API для внешних сервисов, SQL-проверки, прогресса и валидаторов.
- `lms_project/apps/common` - общие модели, включая очередь и историю проверок.

2.2. SQLChecker

FastAPI-сервис для проверки SQL-заданий. Отвечает за:

- выполнение эталонного SQL-запроса и запроса студента;
- сравнение результатов;
- проверку DDL/DML/SELECT-заданий;
- работу с PostgreSQL и ClickHouse;
- статический анализ SQL;
- пакетную и асинхронную проверку;
- health-check и статистику.

Ключевые каталоги:

- `app/api/routes.py` - HTTP API сервиса.
- `app/core/checker.py` - ядро проверки SQL.
- `app/models/schemas.py` - схемы запросов и ответов.
- `tests` - тесты для PostgreSQL, ClickHouse, CTE, оконных функций, JOIN/GROUP BY и других SQL-конструкций.

2.3. code_checker

Сервис проверки программного кода. Выполняет решения студентов в Docker-песочницах и возвращает результат в LMS.

Отвечает за:

- получение задач из очереди LMS;
- запуск кода в изолированном окружении;
- проверку по тест-кейсам;
- проверку задач stdin/stdout;
- запуск пользовательского кода с произвольным вводом;
- статический анализ кода;
- поддержку разных языков и окружений.

Ключевые каталоги:

- `app/worker.py` - воркер, который забирает задания из LMS и отправляет результаты обратно.
- `app/main_checker.py` - общая логика выбора режима проверки и запуска runner'ов.
- `app/runners` - runner'ы для отдельных языков.
- `app/static_analyzers` - статические анализаторы кода.
- `sandbox_images` - Dockerfile'ы песочниц.
- `tests` - unit- и integration-тесты.

3. Общая архитектура

Система построена как набор связанных сервисов.

LMS является центральным приложением. В нем пользователь работает с курсом, отправляет решения, получает результаты и прогресс. Внешние checkers не хранят основной учебный контент, а только выполняют проверки.

Основной поток для SQL-заданий:

1. Студент открывает SQL-шаг в LMS.
2. LMS отправляет SQL-запрос на внутренний endpoint `api/sql/execute/` или `api/sql/validate/`.
3. LMS формирует запрос к внешнему `SQLChecker`.
4. `SQLChecker` выполняет эталонный запрос и запрос студента в изолированной временной БД.
5. Результат возвращается в LMS.
6. LMS обновляет прогресс студента и отображает результат.

Основной поток для programming-заданий:

1. Студент отправляет код в LMS.
2. LMS создает запись в `submission_queue` со статусом `pending`.
3. `code_checker` воркер периодически запрашивает pending-задания через API LMS.
4. Воркер получает payload задания.
5. `code_checker` запускает проверку в Docker-песочнице.
6. Воркер отправляет результат обратно в LMS.
7. LMS переносит запись из очереди в историю и показывает результат студенту.

4. Жизненный цикл учебного контента

Базовая структура контента:

- курс;
- модуль;
- урок;
- шаг.

Шаг может быть разных типов:

- текст;
- видео;
- тест;
- SQL-задание;
- programming-задание;
- свободный ответ;
- вебинар;
- Jupyter.

Для студента LMS отслеживает:

- факт просмотра;
- количество попыток;
- статус выполнения;
- набранные баллы;
- дату завершения;
- историю отправленных решений.

5. SQL-задания

SQL-задание хранится в LMS как отдельный тип шага. Студент вводит SQL-запрос в редакторе, после чего может выполнить запрос или отправить решение на проверку.

Основные endpoints LMS:

- `POST /api/sql/execute/` - выполнение SQL-запроса.
- `POST /api/sql/validate/` - проверка SQL-решения.

Внешний SQLChecker предоставляет:

- `POST /check` - синхронная проверка;
- `POST /check/batch` - пакетная проверка;
- `POST /check/async` - асинхронная проверка;
- `GET /tasks` - список задач;
- `GET /tasks/{task_id}` - статус задачи;
- `GET /health` - состояние сервиса;
- `GET /stats` - статистика;
- `GET /analyze` - статический анализ SQL;
- `POST /template/create` - создание шаблонной БД.

SQLChecker автоматически определяет тип запроса:

- `SELECT` и `WITH` проверяются сравнением результата;
- `CREATE`, `DROP`, `ALTER`, `TRUNCATE` обрабатываются как DDL;
- `INSERT`, `UPDATE`, `DELETE` обрабатываются как DML.

Поддерживаемые СУБД:

- PostgreSQL;
- ClickHouse.

Важные особенности:

- для каждой проверки создается временная база или изолированное окружение;
- эталонный и студенческий запросы выполняются отдельно;
- результат сравнивается с учетом настроек проверки;
- при необходимости можно проверять имена колонок;
- для сложных SQL-конструкций есть тесты: CTE, JOIN/GROUP BY, подзапросы, оконные функции.

6. Programming-задания

Programming-задания проверяются через `code_checker`.

LMS формирует payload и сохраняет его в `submission_queue`. Payload содержит:

- `type`;
- `step_id`;
- `user_id`;
- `mode`;
- `student_code`;

- `code` ;
- `language` ;
- `environment` ;
- `test_cases` ;
- `validator` ;
- `method_name` ;
- `timeout` ;
- `memory_limit` ;
- `points` .

Режим проверки определяется LMS по тест-кейсам:

- если первый `input` в `test_cases` является строкой, используется режим `stdin_stdout` ;
- иначе используется режим `run_tests` ;
- если тест-кейсов нет, по умолчанию используется `run_tests` .

6.1. Режим run_tests

Используется для задач, где решение представлено функцией или методом.

Пример логики:

- автор задает `method_name` ;
- автор задает список тест-кейсов;
- code_checker генерирует тестовый runner;
- runner вызывает функцию/метод студента;
- фактический результат сравнивается с ожидаемым.

6.2. Режим stdin_stdout

Используется для задач классического олимпиадного формата:

- программа читает данные из stdin;
- программа печатает ответ в stdout;
- проверка сравнивает stdout с ожидаемым выводом.

Особенность этого режима: `method_name` не требуется. Это важно, потому что задача проверяет всю программу, а не отдельную функцию.

В `code_checker` режим поддерживается через методы runner'ов:

- `prepare_stdin_stdout_files` ;
- `get_stdin_stdout_command` ;
- `parse_stdin_stdout_results` .

На момент подготовки документации отдельные тесты stdin/stdout есть для Python и Scala.

6.3. Режим run_custom_input

Используется для пробного запуска кода с пользовательским вводом. Это не полноценная проверка всех тестов, а запуск решения с конкретными входными данными.

7. Очередь проверок programming-заданий

Для асинхронной проверки используется таблица `submission_queue` .

Основные статусы:

- `pending` - задание ожидает обработки;
- `processing` - задание взято в работу;
- `checked` - результат получен;
- `failed` - проверка завершилась ошибкой.

После завершения проверки результат переносится в `submission_history` .

Основные модели:

- `SubmissionQueue` - очередь активных заданий;
- `SubmissionHistory` - история завершенных проверок.

Внешний воркер `code_checker` работает по pull-модели:

1. Делает `GET /api/validator/submissions/?status=pending` .
2. Получает список ID задач.
3. Для каждой задачи делает `GET /api/validator/submissions/{id}/` .
4. Выполняет проверку.
5. Отправляет результат через `PATCH /api/validator/submissions/{id}/` .

Для доступа используется заголовок:

```
X-API-Key: <token>
```

Тип валидатора определяется по переменным окружения вида:

```
VALIDATOR_PROGRAMMING_TOKEN=...
VALIDATOR_SQL_TOKEN=...
```

8. Docker-песочницы code_checker

`code_checker` выполняет пользовательский код в Docker-контейнерах. Это нужно для изоляции решений студентов и ограничения доступа к окружению.

Поддерживаемые профили зависят от Dockerfile'ов в `sandbox_images` .

Примеры окружений:

- Python base;
- Python PySpark;
- Python sklearn;
- Python TensorFlow;
- Python PyTorch;
- Java base;
- Java Spark;
- Scala Spark;
- C++;
- C#;
- Go;
- R;
- Rust;
- Haskell.

Для каждого языка используется отдельный runner, который отвечает за:

- генерацию файлов для запуска;
- формирование команды запуска;
- парсинг stdout/stderr;
- преобразование результата к единому формату.

9. Основные переменные окружения

9.1. LMS

Ключевые настройки:

```
DB_USER=...
DB_PASSWORD=...
DB_HOST=...
DB_PORT=...

S3_ACCESS_KEY=...
S3_SECRET_KEY=...
S3_BUCKET_NAME=...
S3_ENDPOINT=...

VALIDATOR_PROGRAMMING_TOKEN=...
VALIDATOR_SQL_TOKEN=...
JITSI_SERVER_URL=...
JITSI_APP_ID=...
JITSI_APP_SECRET=...
JUPYTERHUB_URL=...
```

Также в настройках LMS используются:

- Redis для Celery;
- S3/MinIO для файлов;
- FFmpeg/FFprobe для обработки видео;
- `SQL_CHECKER_URL` для связи с SQLChecker.

9.2. code_checker

```
LMS_BASE_URL=...
LMS_API_TOKEN=...
```

`LMS_BASE_URL` должен указывать на базовый URL LMS. `LMS_API_TOKEN` должен совпадать с соответствующим `VALIDATOR_*_TOKEN` в LMS.

9.3. SQLChecker

```
DB_HOST=...
DB_PORT=5432
DB_USER=...
DB_PASSWORD=...
DB_NAME=...

CH_HOST=...
CH_PORT=8123
CH_USER=...
CH_PASSWORD=...
```

PostgreSQL является обязательным для основной проверки. ClickHouse используется для ClickHouse-заданий.

10. Запуск системы

порядок запуска:

1. Запустить инфраструктуру LMS: БД, Redis, S3/MinIO.
2. Применить миграции LMS.
3. Запустить Django/Gunicorn.
4. Запустить Celery worker и Celery beat, если используются фоновые задачи.
5. Запустить SQLChecker.
6. Собрать Docker-образы `code_checker`.
7. Запустить worker `code_checker`.
8. Проверить health endpoints и тестовую отправку задания.

Для разработки каждая часть может запускаться отдельно. При этом важно, чтобы URL и токены между LMS и внешними сервисами совпадали.

11. Тестирование

11.1. LMS

Для LMS важно проверять:

- создание и редактирование курсов;

- прохождение уроков;
- отправку SQL-заданий;
- отправку programming-заданий;
- обновление прогресса;
- работу очереди и истории проверок.

11.2. SQLChecker

В `SQLChecker/tests` покрываются:

- базовые PostgreSQL-проверки;
- базовые ClickHouse-проверки;
- CTE;
- JOIN/GROUP BY;
- подзапросы;
- оконные функции;
- общие SQL-конструкции.

11.3. code_checker

В `code_checker/tests` есть:

- unit-тесты runner'ов;
- integration-тесты запуска в песочнице;
- тесты статического анализа;
- тесты stdin/stdout режима для Python;
- тесты stdin/stdout режима для Scala;
- тесты API-сервера.

12. Минимальный чек-лист приемки документации

Документация считается достаточной, если по ней можно понять:

- из каких сервисов состоит проект;
- какой сервис за что отвечает;
- как проходит SQL-проверка;
- как проходит programming-проверка;
- чем отличаются `run_tests` и `stdin_stdout`;
- где хранится очередь проверок;

- как внешний worker забирает и возвращает задания;
- какие переменные окружения критичны;
- какие ограничения у текущей реализации.

13. Изменения относительно старой документации

Часть существующей документации в отдельных README-файлах устарела. При подготовке и сопровождении документации нужно ориентироваться на текущее состояние кода.

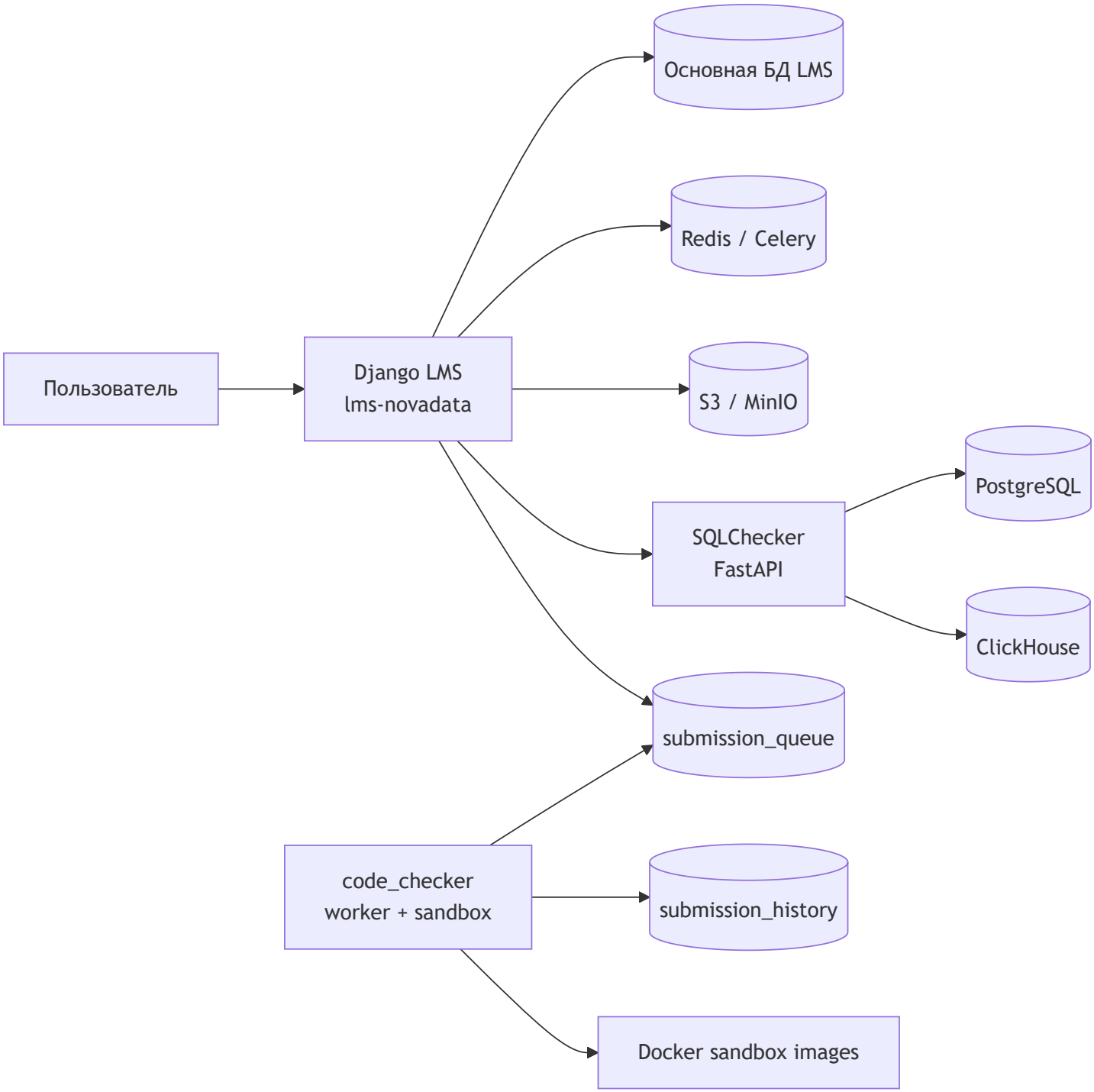
Ключевые актуальные изменения:

- Programming-задания теперь поддерживают отдельный режим `stdin_stdout`.
- LMS автоматически выбирает режим проверки programming-задания по формату первого тест-кейса.
- Если `input` первого тест-кейса является строкой, используется `stdin_stdout`.
- Если `input` задан как структура аргументов функции, используется `run_tests`.
- Для `stdin_stdout` не требуется `method_name`, потому что проверяется вся программа, а не отдельная функция.
- Payload programming-задания содержит и `student_code`, и `code` для совместимости с разными частями checker'a.
- Проверка programming-заданий работает через очередь `submission_queue` и историю `submission_history`.
- Внешний `code_checker` работает по pull-модели: сам забирает pending-задания из LMS.
- Результат проверки возвращается в LMS через `PATCH /api/validator/submissions/{id}/`.
- SQLChecker поддерживает автоопределение типа SQL-запроса: SELECT/DDL/DML.
- Для SQLChecker актуальны тесты PostgreSQL и ClickHouse, включая CTE, подзапросы, оконные функции и JOIN/GROUP BY.

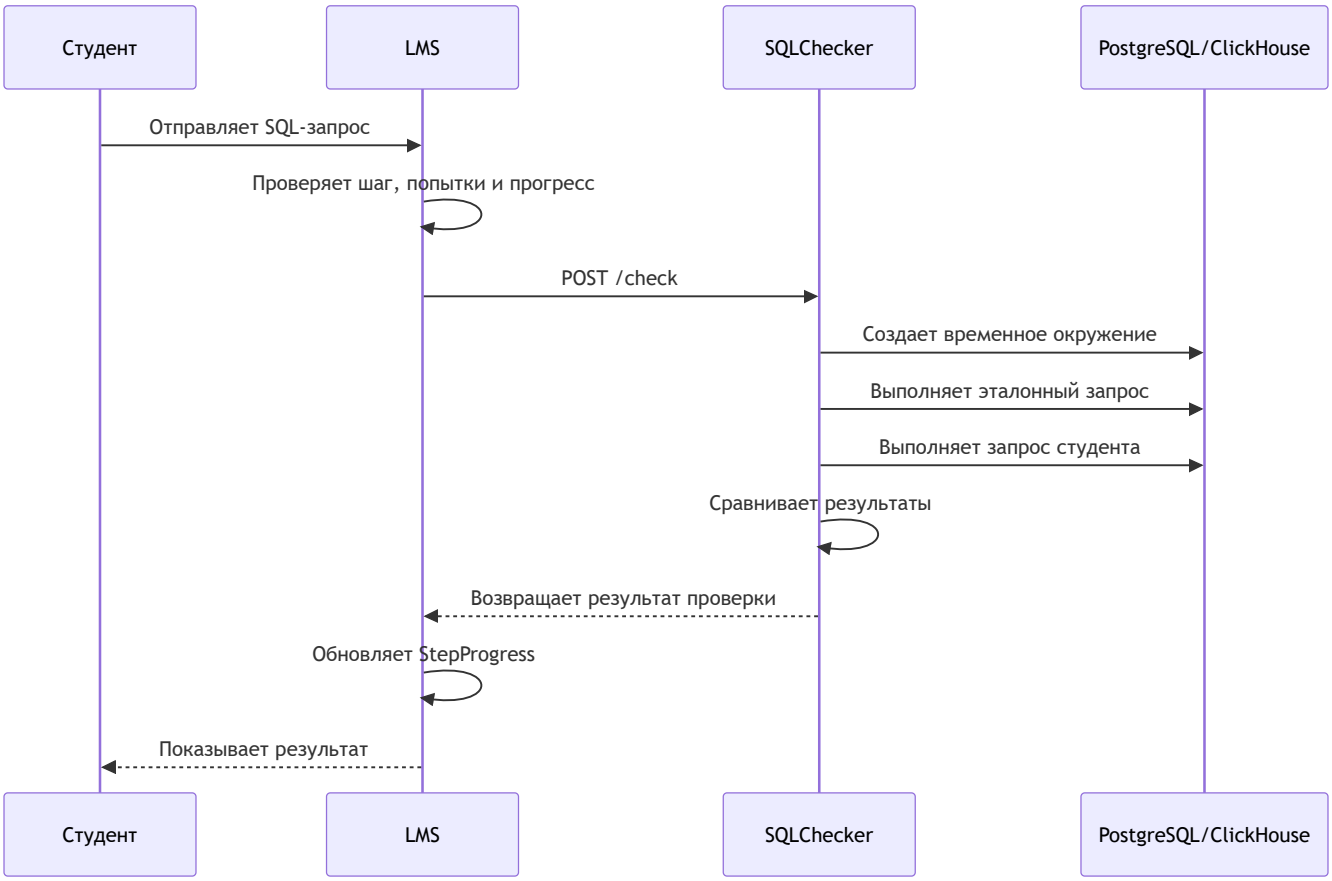
Практический вывод: старые README можно использовать как вспомогательный материал, но перед включением информации в финальную документацию ее нужно сверять с кодом.

14. Схемы взаимодействия сервисов

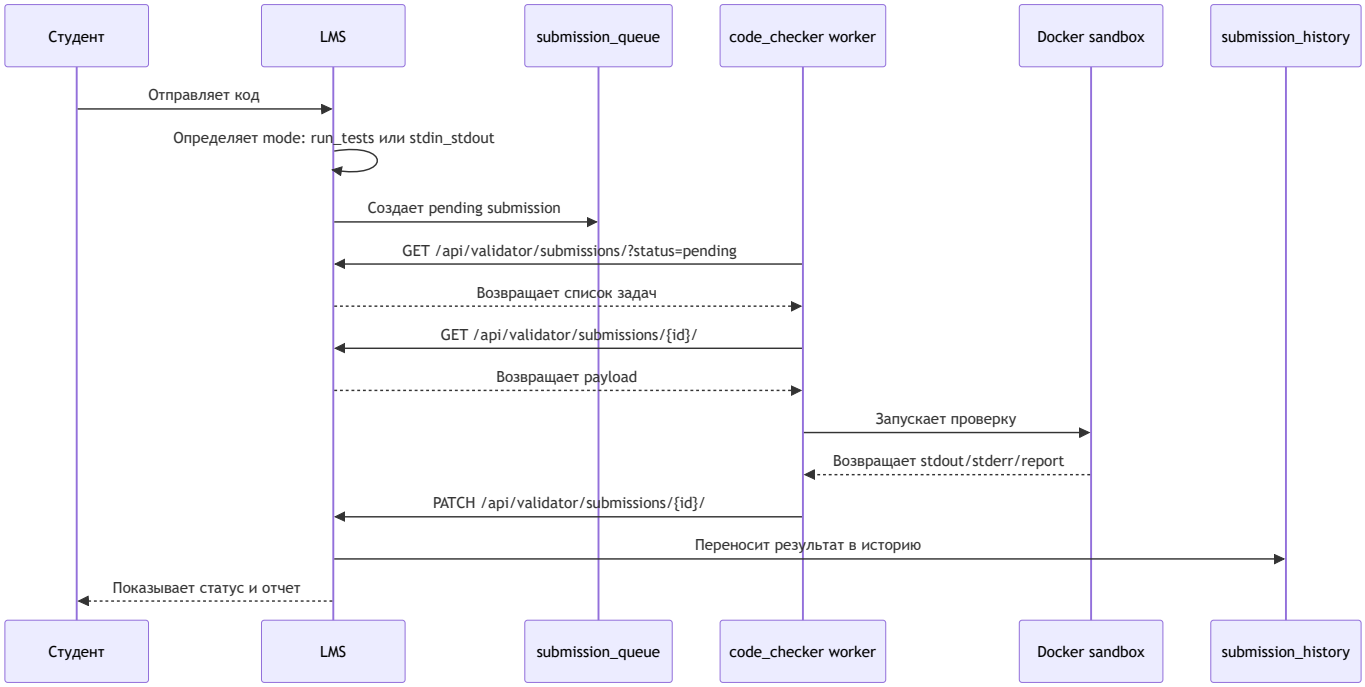
14.1. Общая схема проекта



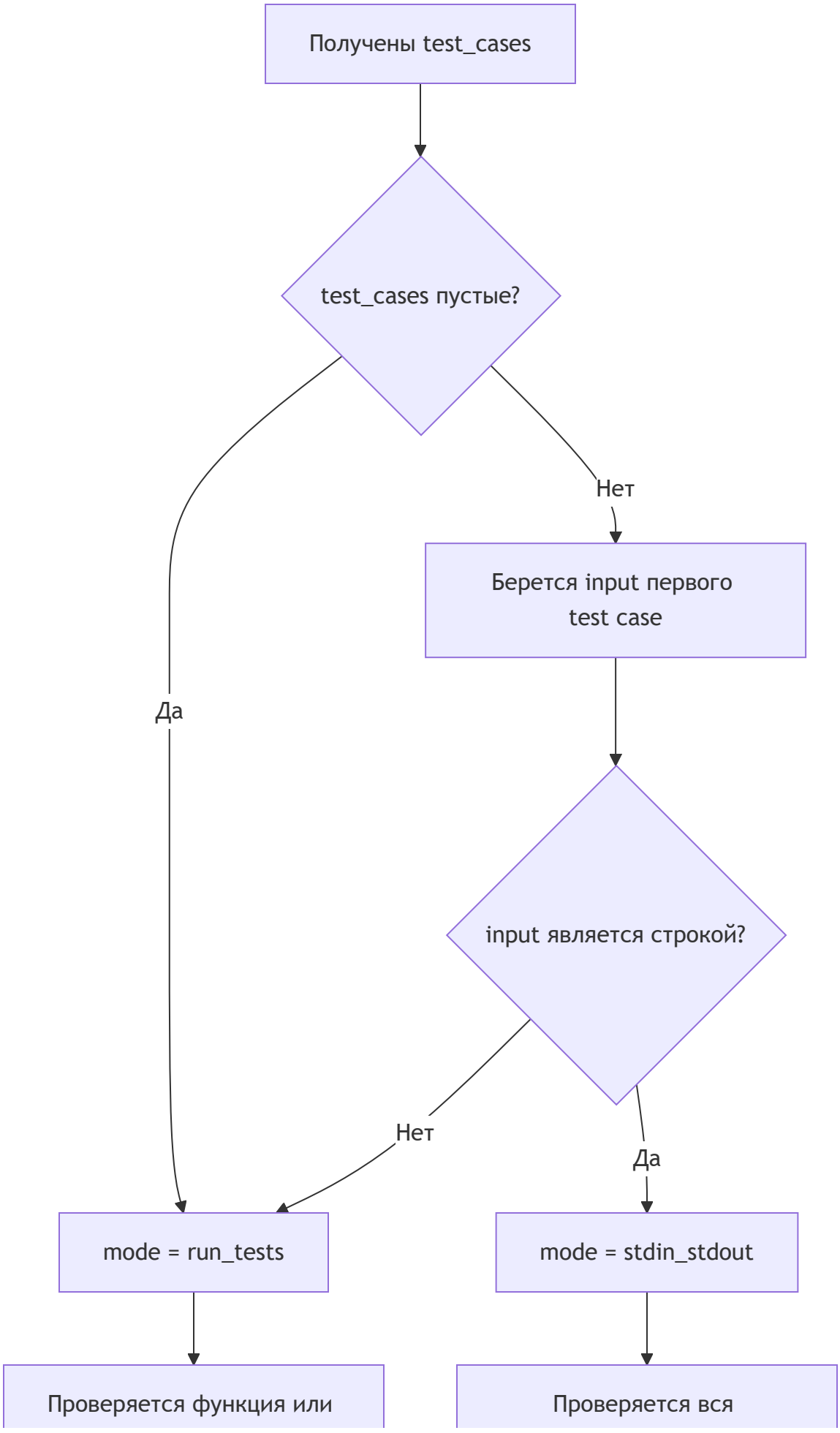
14.2. Проверка SQL-задания

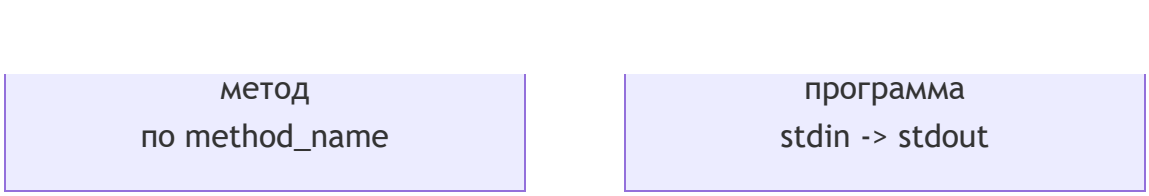


14.3. Проверка programming-задания



14.4. Выбор режима programming-проверки


Получены test_cases
test_cases пустые?
Да
Нет
Берется input первого test case
input является строкой?
Нет
Да
mode = run_tests
mode = stdin_stdout
Проверяется функция или
Проверяется вся


метод
по method_name
программа
stdin -> stdout