ChatGPT

Аудит архитектуры и логики проекта ТОТ – Твоя Точка Опоры

Обзор структуры проекта и кода

ТОТ (Твоя Точка Опоры) – это MVP системы для вызова врачей на дом с микросервисной архитектурой [1]. Backend состоит из отдельных сервисов (Python FastAPI) и API Gateway, а frontend представлен React-приложениями для пациентов и админов [2] [3]. Ниже перечислены ключевые компоненты backend:

- **API Gateway (порт 8000)** – единая точка входа, проксирует запросы к сервисам и проверяет JWT-токены [4] [5]. Gateway реализует маршруты аутентификации, объединённые эндпоинты для админ-панели и прокси для всех доменных сервисов. В коде настроена передача идентификатора пользователя и роли в заголовках (`X-User-ID`, `X-User-Role`) для downstream-сервисов [6] [7]. Для админ-запросов реализована проверка роли: например, эндпоинты `/admin/...` и `/api/...` отвечают **403 Forbidden**, если роль токена не `"admin"` [8].

- **User Service (порт 8001)** – сервис управления пользователями и JWT-аутентификации [9]. Хранит учетные записи, включает регистрацию, логин, выдачу токена и профильные поля пользователя. Модель пользователя содержит UUID в качестве `id` (строка) и поле `role` для типа учетной записи (`patient` по умолчанию, может быть `doctor`, `clinic` или `admin`) [10]. Для докторов и клиник в таблице `users` присутствуют дополнительные колонки (специализация, номер лицензии, стаж, название клиники и т.д.) [11], хотя эти же сведения частично дублируются в Profile Service (см. ниже). Сервис выдает JWT-токен при регистрации/логине, включая `user_id` и роль в payload [12] [13]. Валидация токена при обращениях происходит на уровне Gateway, но User Service также может возвращать данные текущего пользователя по токену (`/auth/me`) [14] [15].

- **Profile Service (порт 8002)** – сервис профилей пациентов, врачей и клиник [16]. Реализует отдельные таблицы для подробных анкет: `profiles` (пациент) с полями мед.данных (дата рождения, группа крови, страховой полис, аллергии и т.п.) [17] [18], `doctor_profiles` (специализация, образование, языки, цена консультации, рейтинг и др.) [19] [20] и `clinic_profiles` (информация о клинике) [21]. Каждый профиль связывается с пользователем по `user_id`. **Важно:** в Profile Service тип `user_id` определён как Integer [22], тогда как User Service генерирует UUID-строки для идентификаторов пользователей [10] – это несоответствие, о котором подробнее ниже. Сервис позволяет создавать и обновлять записи профиля. Однако авторизация тут упрощена: например, при обновлении профиля проверяется, что `user_id` в JWT (в header) совпадает с обновляемым, иначе 403 [23] [24]. В коде Profile Service есть заглушка для извлечения пользователя из заголовка Authorization, ожидающая `Bearer <user_id>` и просто парсящая число [25] – это временное решение вместо полноценной проверки JWT, поскольку Gateway на самом деле передает ID через `X-User-ID`, а не как Bearer-токен.

- **Booking Service (порт 8003)** – сервис управления заказами на визит врача [16]. Содержит основную бизнес-логику приема запросов пациентов, назначения докторов и отслеживания статусов вызова. Модель `Booking` хранит информацию о вызове: пациент, назначенный врач (может быть null до назначения), адрес, симптомы, тип вызова (срочный/плановый/консультация), временные метки и т.д. [26] [27]. Статус заказа (`pending`, `assigned`, `in_progress`, `completed`, `cancelled`) обновляется по ходу процесса [28] [29]. Для отслеживания изменений статуса есть связанная таблица `booking_status_updates` [30]. Реализована также связь `BookingMessage` для чата между пациентом и доктором в рамках заказа [31]. Booking Service принимает запросы через Gateway, используя заголовки `X-User-ID` и `X-User-Role` для авторизации. Например, при создании нового заказа сервис берет `patient_id` из `X-User-ID` заголовка [32] [33]; при получении или отмене заказа проверяется, что либо запрашивающий – участник этого заказа, либо админ, иначе возврат 403 [34] [35]. Сервис также предоставляет эндпоинты для назначения врача на вызов, старта и завершения обслуживания, отправки сообщений в чат и т.д., о чем ниже в описании процессов.

- **Payment Service (порт 8005)** – сервис платежей и внутреннего кошелька [36]. Отвечает за создание платежей через **ЮKassa** или **СБП** (Система быстрых платежей), а также учет внутренних балансов пользователей. Модель `Payment` хранит платежи (ID, пользователь, сумма, метод, статус и т.д.) [37], `Wallet` – баланс пользователя [38], `Transaction` – операции по кошельку (пополнение, списание) [39]. Для интеграции с ЮKassa подключён SDK (если настроен ID магазина и секретный ключ) [40], а для СБП реализована генерация QR-кода оплаты [41] [42]. При создании нового платежа сервис генерирует `transaction_id` и сохраняет запись со статусом “pending” [43] [44], после чего, в зависимости от метода, либо делает запрос в ЮKassa (получая данные для переадресации) [45] [46], либо формирует QR-код для СБП [47], либо сразу списывает со внутреннего кошелька [48]. **Обратите внимание:** как и в Profile Service, авторизация здесь упрощена – для некоторых операций ожидается заголовок Authorization с числовым user_id [49]. Gateway же не передаёт Authorization в Payment Service (только X-User-ID), поэтому методы, требующие `current_user_id: int = Depends(get_user_from_header)`, могут не работать из-под Gateway. Например, эндпоинт получения всех платежей пользователя (`GET /payments/user/{user_id}`) проверяет, что `user_id` совпадает с токеном [50], но токен не будет распознан без доработки. В текущем MVP эти нюансы авторизации на уровне Payment Service не критичны, т.к. основные вызовы (создание и получение платежа) не требуют проверки текущего пользователя (их может вызывать Gateway напрямую).

Зависимости и стиль кода: Все сервисы используют FastAPI и SQLAlchemy (с SQLite по умолчанию) для хранения данных [51] [52]. Код каждого сервиса сконцентрирован в одном файле `main.py`, что упрощает обзор, но ведет к очень большим модулям. Логирование включено на уровне INFO [53], хотя в бизнес-логике детальных логов немного (в основном при ошибках). Авторизация JWT организована через общую секретную фразу и проверяется на Gateway (вместо единого Auth-сервиса) – это допустимо для MVP, но вызывает дублирование кода для проверки токена в каждом сервисе (где-то реализовано полноценно, где-то временная заглушка).

В целом архитектура **логически разделена по доменам** (пользователи, профили, заказы, платежи и т.д.), что повышает модульность. Межсервисное взаимодействие реализовано синхронно через http-запросы от Gateway (используется httpx AsyncClient) [54] [55]. Такой подход прост, но приводит к сильной зависимости Gateway от адресов микросервисов (они заданы как ENV-переменные или дефолты в словаре SERVICES [5]).

Стоит отметить, что **frontend** (React) обращается к тому же API Gateway. Для локальной разработки все сервисы и приложения запускаются скриптами PowerShell/шела [56], можно запускать всю систему или отдельные части. Демо-аккаунты администратора и пациента указаны прямо в README [57].

Итог по структуре: Проект охватывает основные компоненты для MVP: регистрацию пользователей, профильную информацию, процесс бронирования врача, чат в рамках визита, оплату услуг и админ-панель для модерации. Далее подробно рассмотрены бизнес-процессы и их реализации в коде, с указанием сильных и слабых сторон каждого.

Бизнес-процессы: схемы и оценка логики

Ниже приведена декомпозиция основных сценариев использования системы ТОТ. Для каждого бизнес-процесса дано описание шагов (в форме приближенной блок-схемы) и анализ завершённости реализации в текущей версии.

1. Регистрация и аутентификация пользователей

Описание процесса: Новый пользователь (пациент или врач/клиника) должен иметь возможность создать аккаунт и войти в систему. Администратор как правило добавляется напрямую в БД или через скрипт, но для MVP можно регистрировать и админов переключением роли. Аутентификация основана на JWT токенах.

Flow регистрации/логина:

1. **[Подача данных]:** Пользователь заполняет форму регистрации (email, телефон, имя, пароль и т.д.) и отправляет запрос `POST /auth/register` через Gateway.
2. **[Проверка уникальности]:** User Service получает запрос `/auth/register`. Он проверяет, нет ли уже пользователя с таким email или телефоном [58] [59]. Если существует, возвращается ошибка **400 Bad Request** ("User already exists") [60].
3. **[Создание учетной записи]:** Если данные уникальны, создается новая запись User: поля заполняются, пароль хэшируется (bcrypt) [61] [62]. Роль по умолчанию – "patient", если не указано иначе (но админ или врач могут задать другое значение поля role на этапе регистрации) [63] [64]. Все дополнительные поля (специализация, лицензия и т.д.) сохраняются, что позволяет сразу регистрировать врача или клинику, хотя их профиль потребуется дополнить.
4. **[Выдача JWT]:** После сохранения нового пользователя User Service генерирует JWT-токен: в payload включаются `user_id` (UUID) и роль пользователя [12] [13]. Время жизни токена берется из настройки (по умолчанию 24 часа) – в коде изначально использовался `datetime.now()`, что могло приводить к рассинхронизации времени, но в последнем обновлении это исправлено на UTC [65] [66]. Токен возвращается клиенту вместе с основными данными пользователя (в ответе `TokenResponse`) [67] [68].
5. **[Аутентификация (логин)]:** При последующем входе `POST /auth/login` пользователь указывает email и пароль. User Service находит запись по email [69], проверяет пароль через bcrypt [70]. В случае успеха и если учетная запись не деактивирована (`is_active=True`) [71], выдается новый JWT по тому же алгоритму [12]. В противном случае – ошибка **401**.
6. **[Доступ к ресурсам]:** Получив JWT, клиент включает его в заголовок Authorization для всех API-запросов. Gateway через `Depends(verify_token)` валидирует токен на каждом

защищенном маршруте [72]. Если токен просрочен или неверен – ответ **401 Unauthorized** с соответствующим сообщением [73].

Оценка реализации: Базовый функционал регистрации/логина **реализован полно**: уникальность email/телефона проверяется, пароли хранятся безопасно (хэшируются) [74] [70], JWT генерируется и проверяется. Логика понятна и завершена для MVP.

Несколько улучшений, которые можно отметить:

- **Верификация контактов:** Поле `is_verified` у пользователя всегда `False` после регистрации, и в коде нет процесса подтверждения email/SMS. Для продакшена понадобится механизм верификации (отправка кода или ссылки) – сейчас администратор мог бы вручную менять `is_verified=True` через обновление пользователя, но автоматизации нет.

- **Управление ролями:** Хотя сервис позволяет указать роль при регистрации, в пользовательском приложении, вероятно, регистрируются только пациенты. Создание врача/клиники может делать админ (см. раздел про модерацию), либо врач регистрируется сам как пациент и получает роль **doctor** позднее. Четкая бизнес-логика этого момента не зафиксирована в коде, но поддержка ролей есть.

- **Безопасность JWT:** В свежей версии проблемы с временем жизни токена решены (везде UTC, токен живет 72 часа в тестовом режиме) [65]. Можно добавить *refresh token* для продления сессии, но для MVP это не критично.

В целом, процесс аутентификации готов для MVP. Пользователь может зарегистрироваться и сразу использовать полученный токен для доступа к функционалу (личный кабинет, запись к врачу и т.д.) через API Gateway.

2. Профиль пользователя и обновление данных

Описание процесса: После регистрации пациенту желательно заполнить персональный профиль (например, добавить адрес, дату рождения, мед. данные), а врачу – информацию о себе (образование, лицензии). Эти данные хранятся в Profile Service. Разделение на User Service и Profile Service позволяет не засорять основную таблицу пользователей деталями профиля и разграничить доступ (например, админ может редактировать профили врачей, не затрагивая учетные записи).

Flow управления профилем:

- **Создание профиля пациента:** При первом входе пациент может заполнить анкету. Клиентское приложение должно отправить `POST /profiles/` (через Gateway, который на данный момент *не* проксирует этот маршрут, то есть вызов должен идти прямо на Profile Service либо потребовалась бы настройка Gateway). Параметры – `user_id` (идентификатор пользователя, полученный при регистрации) и поля профиля (телефон, адрес, дата рождения, мед.история и т.д.) [75] [76]. Profile Service создает запись в таблице `profiles` [77] [78]. **Примечание:** В MVP создание профиля не инициируется автоматически – нет триггера из User Service. Это означает, что профиль пациента может остаться пустым, пока пользователь сам не внесет данные через UI.

- **Обновление профиля:** Для изменения данных пациент отправляет `PUT /profiles/{user_id}` с нужными полями. Gateway проксирует на Profile Service, добавляя заголовок `X-User-ID`. В Profile Service эндпоинт `/profiles/{user_id}` проверит, что авторизованный `current_user_id` (из заголовка Authorization, который в текущей реализации не приходит, поэтому фактически сравнение идет с None) совпадает с запрашиваемым user_id [23]. Если всё в порядке, профиль находится и обновляются переданные поля [79] [80]. Сервис ограничивает обновление **только владельцем** – чужой профиль нельзя изменить (кроме администратора, который мог бы вызвать Profile Service напрямую без этого механизма). После сохранения возвращаются обновленные данные [81].

- **Профиль врача и клиники:** Эти профили обычно создаются/редактируются администратором. Процесс схож: `POST /doctor-profiles/` сохраняет запись для врача (ожидая `user_id` существующего пользователя-врача) [82]; `PUT /doctor-profiles/{user_id}` обновляет данные врача, но требует соответствия текущего пользователя – на практике, врач может обновлять **свой** профиль, а админ мог бы отправить запрос, обойдя проверку (Gateway для админ-запросов пока не настроен на эти эндпоинты). Профиль клиники (`/clinic-profiles/`) работает аналогично [83] [84]. Админ-панель вызывает их через `/api/doctors` и `/api/clinics` маршруты Gateway, которые перенаправляют на соответствующие эндпоинты Profile Service [85] [86].

Оценка реализации: Хранение профилей вынесено в отдельный сервис, что правильно концептуально, но в текущем коде есть **неполадки:**

- Во-первых, **несоответствие типов** `user_id`**:** Profile Service ожидает `user_id` как целое число (видимо, планировалось использовать числовые ID) [22], тогда как User Service использует UUID-строки для идентификаторов пользователей [10]. Это приводит к тому, что `user_id` профиля пациента, например, может быть `None` (если парсинг Bearer-токена не удался) или просто не будет совпадать с реальным UUID пользователя. В результате привязка профиля к пользователю выполняется **на уровне приложения** (т.е. клиент знает свой user_id и передает его), но целостность на уровне базы не гарантируется (нет внешнего ключа на таблицу `users` и разные типы). Для MVP это не ломает работу напрямую, но чревато несогласованностью данных. Решение – перейти на единый тип ID (например, хранить user_id как строку и в профилях).

- **Дублирование данных врача:** как отмечалось, есть повторяющиеся поля – например, `specialization`, `license_number`, `experience_years` хранятся *и* в User Service (в таблице users) [11], *и* в DoctorProfile [19]. Сейчас при регистрации врача эти поля заполняются в User (чтобы сразу что-то было), но в Profile Service они изначально пусты. Без синхронизации может получиться ситуация, что часть данных обновится в одном месте, а в другом – устареет. Для разруливания можно либо убрать из User эти колонки (оставив только в профиле), либо заполнить профиль сразу при регистрации врача (что не реализовано).

- **Создание профиля по умолчанию:** пациент, зарегистрировавшись, не получает автоматически запись в `profiles`. То есть при попытке получить `/profiles/{user_id}` сразу после регистрации будет **404** (“Profile not found”) [87] [78]. В админ-панели, вероятно, это учтено (список пациентов может формироваться по списку users с ролью `patient` через User Service, минуя профиль). Но для консистентности можно было бы создавать пустой профиль при регистрации или при первом обращении.

- **Авторизация и доступ:** В текущем коде Profile Service доверяет идентификатору пользователя из заголовка Authorization (`get_user_from_header`). Gateway же отправляет ID в `X-User-ID`, а Authorization использует только для User Service. Из-за этого проверка `if user_id != current_user_id` в Profile Service не будет работать правильно (она всегда падает на 401, потому что `authorization` header не содержит ожидаемых данных) [25]. На практике, пока Gateway не проксирует `Authorization` сюда, профили может безопасно менять только админ напрямую (или если изменить Profile Service, чтобы он использовал `X-User-ID`). Это явный **баг MVP**, снижающий работоспособность процесса обновления профиля пользователем.

Вывод: Функциональность профилей задумана и частично реализована (сохранение и получение данных работает), но требует **доработки в части согласованности и авторизации**. Для MVP можно считать, что админ сможет управлять профилями врачей/клиник через свои эндпоинты, а пациент – в теории через Profile Service (хотя из-за бага может не получиться без правки кода). В продуктивной версии необходимо унифицировать идентификаторы, устранить дублирование полей и настроить корректную проверку JWT в Profile Service (либо делегировать эти задачи Security-сервису, если планируется).

3. Оформление вызова врача на дом (Booking Lifecycle)

Описание процесса: Это ядро системы – пациент создает заказ на вызов врача, заявка обрабатывается, доктор выезжает и оказывает услугу, затем визит завершается. Процесс включает несколько ролей (пациент, админ/диспетчер, врач) и ряд возможных состояний заказа. В MVP не реализовано автоматическое распределение врачей, поэтому предполагается, что либо администратор вручную назначает доктора, либо пациент выбирает конкретного врача (но в коде выбор врача пациентом не прослеживается, скорее назначение выполняется отдельно). Также в рамках визита есть возможность переписки (чат) и требуется оплата – эти части рассмотрим в отдельных разделах.

Flow процесса вызова:

1. **[Новая заявка от пациента]:** Авторизованный пациент на клиенте заполняет форму вызова: выбирает тип вызова (например, плановый визит или срочный), описывает симптомы, указывает адрес, предпочтительное время и т.д. Далее отправляет запрос `POST /bookings` на Gateway. Gateway проверяет JWT (роль может быть patient) и пересылает запрос в Booking Service, добавляя `X-User-ID` равный ID пациента [88].
   Booking Service создает новый объект `Booking`: проставляет `patient_id` = ID пациента из заголовка, остальные поля берет из запроса [89] [90]. На этапе создания **доктор еще не назначен**, поле `doctor_id` остается `NULL` (в таблице) и статус заявки – `"pending"` (значение по умолчанию) [28] [91]. Заявке присваивается уникальный UUID `id`. Одновременно фиксируется событие статуса: создается запись `BookingStatusUpdate` с пометкой, что заявка создана (старого статуса не было, новый – pending) [91]. После сохранения сервис возвращает клиенту основные данные заказа (в соответствии с моделью `BookingResponse`) [92] [93].

2. **[Обработка заявки и назначение доктора]:** Поступившая заявка должна быть кем-то принята. В текущей реализации автоматического мэтчинга нет: предполагается, что администратор в админ-панели увидит новую запись (статистика или список открытых заказов). Админ может связаться с подходящим врачом вне системы и определить, кто возьмет вызов. После этого админ назначает врача через эндпоинт `PUT /bookings/`

`{booking_id}/assign?doctor_id=...`. **Gateway** имеет маршрут `/api/appointments` (упомянут в документации) или может напрямую проксировать на `/bookings/.../assign`. В коде Gateway **в версии на гитхабе отсутствует** явный `/api/appointments`, но назначение возможно вызовом `/bookings/{id}/assign` от имени админа (JWT с ролью admin). Booking Service проверит роль: для назначения разрешены только **admin или system** (системные сервисы) [94] [95]. Запрос содержит идентификатор врача (вероятно, UUID пользователя-врача). Сервис найдет запись Booking по id и убедится, что она еще в статусе `pending` [96]. Затем проставит `doctor_id` и изменит статус на `assigned` [29] [97], зафиксирует время назначения `assigned_at` и время обновления [98] [99]. Добавляется запись в историю статусов: old_status=pending, new_status=assigned, кто обновил – админ (его user_id) [100] [101]. Результат – ответ **200** с сообщением об успехе [102].

Примечание: Возможна и альтернативная ветка: **выбор врача самим пациентом** при создании заявки. Код Booking Service позволяет в модели Booking иметь `doctor_id`, но Pydantic-модель **BookingCreate** не включает это поле (doctor_id игнорируется при создании) [103] [104]. Значит, пациент через API не назначает конкретного врача – он лишь создает запрос. Таким образом, роль администратора/диспетчера в MVP критична для назначения. В будущем, возможно, добавят поиск ближайших доступных врачей (`GET /geo/doctors/nearby` уже зарезервирован) [105] или уведомление врачей-участников системы через бота. Пока же **назначение врача – ручной процесс** в админ-панели.

1. **[Выполнение вызова врачом]:** Когда врач назначен, ему необходимо приступить к выполнению. Предположительно, врач будет иметь свое мобильное приложение (Flutter, упомянут в контексте) или телеграм-бот. В текущем MVP UI для врача нет, но backend предусматривает действия врача через API:
2. Врач (под своей учетной записью, роль `doctor`) делает запрос `PUT /bookings/{booking_id}/start`. Gateway пропускает (JWT проверен) и шлёт на Booking Service. Сервис убеждается, что `booking.doctor_id` совпадает с `X-User-ID` (т.е. что именно назначенный врач вызывает) [106] [107]. Также проверяет, что статус сейчас `assigned` [108] – только тогда можно “start”. Если всё ок, статус меняется на `in_progress`, фиксируется `started_at = now` [109] [110] и добавляется запись статуса (assigned -> in_progress) [111]. Возвращается подтверждение [112]. Это означает, что врач выехал и приступил к работе.

3. По завершении работы врач вызывает `PUT /bookings/{booking_id}/complete` с данными фактического оказания услуги. В теле `BookingUpdate` могут быть `actual_duration` (сколько минут длился визит), `final_price`, возможно, дополнительные заметки о выполнении [113] [114]. Booking Service проверяет: вызывающий – назначенный врач (как и для start) [115], статус должен быть `in_progress` [116]. Затем присваивает статус `completed`, фиксирует время завершения (`completed_at`) [113] [117], обновляет поля из предоставленных (продолжительность, итоговая цена, заключение) [118] и пишет запись статуса (in_progress -> completed) [119]. В ответе – сообщение об успешном завершении [120]. После этого заказ считается закрытым.

4. **[Отмена вызова]:** На любом этапе до **completed** пациент или админ могут отменить вызов. Например, пациент решил отказаться или админ видит, что нет доступных врачей. Для этого предусмотрен `PUT /bookings/{booking_id}/cancel`. Booking Service принимает запрос и проверяет права: отменять может либо админ, либо сам пациент этого заказа, либо назначенный врач [121] [122]. Чужой пациент или другой врач получит 403. Затем проверяется текущий статус: если уже **completed** или **cancelled**, отмена невозможна [123]. Иначе статус меняется на `cancelled`, фиксируется время отмены

`cancelled_at` [124] [125]. Добавляется запись статуса (от предыдущего состояния к cancelled, с указанием причины если передана) [126]. Клиенту возвращается сообщение об успешной отмене [127]. После отмены заказ закрыт; в дальнейшем можно реализовать возврат денег, уведомления и т.д.

5. **[Просмотр заявок пользователем]:** Пациент может запросить список своих вызовов – эндпоинт `GET /bookings` без ID возвращает заказы текущего пользователя. Booking Service фильтрует по `patient_id` равному `X-User-ID` если роль patient, или по `doctor_id` если роль doctor [128] [129]. Администратор, вызвав этот эндпоинт (через свой токен), получит 403 (для админа нет явного сценария здесь, он пользуется другим маршрутом). Доступна опциональная фильтрация по статусу (query-параметр), ограничение по лимиту и сдвиг для пагинации [130] [131]. Результат – список BookingResponse, отсортированных по дате создания (новые впереди) [132] [133]. Таким образом, пациент в приложении видит историю своих вызовов врача.

Оценка реализации: Основной сценарий **оформления и выполнения вызова** детально проработан в коде. Все важные шаги (создание, назначение, старт, завершение, отмена) имеют соответствующие эндпоинты и проверки:

- Состояния заказа меняются последовательно, невозможны нелогичные переходы (например, нельзя завершить, минуя старт – будет 400 ошибка [116], или назначить врача на уже начатый/завершенный заказ – тоже запрет) [134].
- Права доступа контролируются: пациент не увидит чужой заказ, врач не изменит чужой вызов, админ может выполнять административные действия. В коде явно прописаны проверки ролей и соответствия user_id заказа [35] [122].
- Ведется история статусов (хотя пока нет отдельного API, чтобы эту историю запросить, но можно добавить).

В то же время, на уровне MVP выявляются такие моменты:

- **Ручное назначение врача:** Отсутствует автоматизированный подбор врача. Для старта это приемлемо, но нужно UI для администратора. Судя по changelog, в админ-панели есть страница “Записи” для просмотра всех вызовов [135], однако явных упоминаний интерфейса назначения нет. Возможно, назначение происходит вне системы (по телефону), а админ просто вносит ID врача через базу или будущую функцию. В перспективе стоит реализовать или **уведомление всех врачей** о новом вызове (через бот) или хотя бы интерфейс выбора из списка свободных врачей и кнопку “Назначить”. В коде задел для этого есть (эндпоинт /assign).

- **Отсутствие интерфейса врача:** Несмотря на то, что API для врача реализовано, фактически врач сейчас не имеет приложения для взаимодействия. Это означает, что после назначения заказ зависнет в статусе `assigned`, пока врач не воспользуется API напрямую. Для MVP, возможно, эти шаги будут имитироваться админом (например, админ может и стартовать, и завершить визит от имени врача через специальные сервисные токены). Но в конечной системе нужен либо **Doctor App (Flutter)**, либо хотя бы веб-интерфейс, либо интеграция с Telegram-ботом для уведомления врача и возможности нажать “Начал”/“Завершил”.

- **Чат (переписка) в рамках визита:** В заказ встроена возможность отправлять сообщения (см. следующий раздел), но без активного участия врача через интерфейс эта функция сейчас ограничена. Пациент может написать, но ответит ли кто-то – под вопросом. Тем не

менее, хранение сообщений в базе налажено, и при появлении приложения врача чат заработает.

- **Уведомления и мониторинг:** MVP не включает отдельный Notification Service (хотя в Gateway маршруты зарезервированы) [136]. Это значит, что push-уведомления (например, "Врач выехал" или "Пациент отменил вызов") пока не рассылаются автоматически. Пользователю надо вручную обновлять статус заказа (например, через периодические опросы `/bookings/{id}`) чтобы узнать об изменениях. Для продвинутого UX стоит добавить websocket или бот-оповещение.

- **Работа с геоданными:** Имеются черновые эндпоинты `/geo/doctors/nearby` и `/geo/track` [105] [137]. Они не реализованы (в коде Geo Service нет), но задуманы для поиска ближайших врачей и отслеживания их геолокации. В MVP эти функции неактивны, но архитектура предусматривает их добавление.

В целом процесс **вызова врача** в текущем виде **функционален**, если есть активная роль администратора, компенсирующая недостающий автоматический функционал. Пациент может создать запрос, заявка сохранится; админ сможет увидеть (через прямой запрос к Booking Service или базу) и назначить врача; врач (или админ от его имени) – отметить выезд и завершение. Все данные (включая переписку) сохраняются. Для MVP этого достаточно, однако **для выхода в продакшн** нужно разработать интерфейс для врачей и систему оповещения, иначе процесс будет сильно зависеть от ручного администрирования и коммуникаций вне системы.

4. Обсуждения и чат между пациентом и врачом

Описание процесса: Сервис предполагает возможность общения между пациентом и врачом (или администратором) по конкретному вызову. Это полезно, например, чтобы уточнить детали до визита или дать консультацию. В архитектуре есть два подхода: реализовать отдельный **Chat Service** (см. зарезервированный порт 8007 и маршруты `/chat/rooms` в Gateway) [138] [139], либо встроить чат в контекст конкретного заказа. В MVP выбран второй путь: сообщения хранятся в `BookingMessage` и привязаны к `booking_id` [140]. То есть **каждый заказ имеет свой поток сообщений**, доступный только участникам этого заказа.

Flow общения:

- **Инициация чата:** После создания заказа пациент может перейти в экран чата по этому вызову (например, кнопка "Чат с доктором" на странице заказа). На стороне клиента известен `booking_id`. Пока врач не назначен, общение, вероятно, идет с администратором или ожидание. В коде не запрещено писать даже без назначенного врача – сообщение все равно сохранится с `sender_role="patient"`.

- **Отправка сообщения:** Пользователь отправляет текст; приложение делает запрос `POST /bookings/{booking_id}/messages` на Gateway. Gateway перенаправляет его на Booking Service (маршрут `/bookings/{id}/messages`) [141] [142]. Booking Service проверяет, что отправитель имеет право писать: либо он patient этого заказа, либо назначенный doctor, либо admin [143] [144]. Если чужой – выдаст 403. Затем создается запись `BookingMessage` в базе с полями: уникальный `id`, `booking_id`, `sender_id` (равен `X-User-ID` из заголовка), `sender_role` (роль из заголовка), тип сообщения (`text` по умолчанию) и контент (текст) [145] [146]. `created_at` проставляется автоматически.

Сохраненное сообщение возвращается в ответе (сформировано по `MessageResponse`) [147]. Клиент отображает отправленное сообщение в списке.

- **Получение сообщений:** При открытии чата или для обновления ленты клиент периодически запрашивает `GET /bookings/{booking_id}/messages`. Booking Service на запрос `/messages` также проверяет права доступа (тот же условный допуск: либо пациент, либо врач того же заказа, либо админ) [148] [149]. Затем выбирает из базы все сообщения по данному booking_id, сортируя по времени создания (в коде – последние сообщения первыми, `order_by(... desc())`, что может требовать реверса на клиенте) [150]. Возвращается список сообщений с их полями (ид, отправитель, текст, timestamp) [151]. Клиент отображает их в чате.

- **Роли в чате:** Если администратор хочет участвовать в переписке, он технически может – Gateway не передает явно админа как участника, но при необходимости админ может дернуть Booking Service напрямую. В основном чат рассчитан на пару "пациент-врач". При этом `sender_role` сохраняется, чтобы, например, показывать сообщения разным цветом (пациент vs доктор).

Оценка реализации: Коммуникация пациент-врач через сообщения **прописана**, но страдает от отсутствия интерфейса для врача. Тем не менее:

- **Целостность:** Сообщения привязаны к заказу, и никто вне участников не может их получить или отправить – это верно задано проверками [143] [148]. Администратор при необходимости может просматривать переписку (будучи суперпользователем, он мог бы иметь отдельный сервис для модерации чатов, но сейчас нет).

- **Реализация UI:** На стороне пациента чат уже предусмотрен (раз раздел `/chat` указан в готовых страницах [152]). Скорее всего, пациентское приложение подключается к `/bookings/{id}/messages` через APIService (в changelog упоминается переход на реальные API запросы для admin-panel, вероятно аналогично и для patient-app). Без участия врача, чат односторонний – пациент может написать, но ответить некому до назначения врача. Врач же, не имея приложения, не узнает о сообщении.

- **Потенциальное улучшение:** Планировалось, возможно, интегрировать Telegram-бота для уведомления врача о новых сообщениях или для реализации чата через Телеграм (тем более, в контексте указан **aiogram, FSM**). Пока бот не подключен, можно считать, что чат – задел на будущее. **FSM (Finite State Machine)** могла бы пригодиться именно в боте для ведения диалога (например, бот задает пациенту вопросы последовательно), но внутри текущего кода FSM не используется. В перспективе стоит вынести чат в отдельный сервис с WebSocket для реального времени, или же подключить push-уведомления.

- **Отсутствие мультимедиа:** Пока сообщения – только текст (есть поле `message_type` для потенциала, но загрузка изображений/аудио не реализована). Для телемедицины, возможно, потребуется отправка фото анализов и т.п., но MVP сфокусирован на тексте.

Подытоживая, **функция обсуждений** в MVP скорее концептуальная: архитектура позволяет переписку и сохраняет её, но фактическая полезность появится лишь с внедрением интерфейса для врача. В текущем виде чат можно считать **частично работоспособным** (сообщения сохраняются и выводятся), но не **полностью функциональным** без реальных пользователей с

обеих сторон. Для улучшения коммуникации к продакшену рекомендуется либо интеграция с ботом, либо запуск мобильного приложения врача с уведомлениями о новых сообщениях.

5. Обработка платежей за услуги

Описание процесса: Система ТОТ должна обеспечивать оплату медицинских услуг на дому. MVP включает интеграцию с российскими платежными системами – YooKassa (прием банковских карт, СБП и др через API ЮMoney) и прямую генерацию QR для СБП, а также ведение внутреннего кошелька пользователя (например, для бонусного счета или предоплаты). Бизнес-процесс оплаты связан с процессом бронирования: пациент после заявки должен оплатить визит (либо до, либо после оказания услуги). Судя по структурами данных, поддерживается сценарий **предоплаты** (есть поле `estimated_price` при создании заказа [153] и сразу оплата) или **оплаты по факту** (когда известна `final_price` после завершения визита). Рассмотрим процесс на примере предварительной оплаты:

Flow платежного процесса:

1. **[Инициация оплаты]:** После назначения врача и согласования стоимости услуги, пациент в приложении нажимает “Оплатить”. Ему предлагается выбор способа: **Банковская карта (YooKassa)**, **СБП** или **Внутренний кошелек**. Допустим, выбрана YooKassa. Клиент отправляет запрос `POST /payments/create` (Gateway проксирует на Payment Service `/payments/`). В теле – сумма, валюта, описание, метод (`"yookassa"`), а также `user_id` (и `booking_id` для привязки к заказу) [43] [44].

2. **[Создание записи платежа]:** Payment Service получает запрос и сначала создает запись в БД: генерирует `transaction_id` (UUID) для отслеживания [43]. Создается объект Payment со статусом `"pending"`, указанной суммой, методом и привязкой к пользователю/заказу [154]. Сразу сохраняется в базе и обновляется (чтобы получить ID) [155].

3. **[Обработка через провайдера]:** Далее сервис разветвляет логику по `payment_method`:

4. **Метод YooKassa:** Сервис пытается вызвать SDK YooKassa для создания платежа. Передает сумму, описание, URL для возврата (можно настроить, у них прописан шаблон `https://tot-mvp.ru/payment/return/{transaction_id}`) и метаданные (например, привязка payment_id и user_id) [45] [156]. Если SDK настроен правильно (в .env заданы YUKASSA_SHOP_ID и SECRET_KEY), будет создан платеж в системе YooKassa. От SDK возвращается объект с полями: уникальный id платежа в YooKassa, статус (например, `"pending"` до оплаты), флаг paid (False), ссылка на оплату (confirmation URL) и т.д. Сервис берет эти данные и обновляет запись Payment: проставляет `payment_provider="yookassa"`, `provider_payment_id` = id от YooKassa, `payment_metadata` = JSON с остальными данными [157] [158]. После чего оставляет статус pending и возвращает Payment объект. Клиент, получив ответ, извлекает URL для оплаты и перенаправляет пользователя на платежную страницу YooKassa (или открывает WebView). Пользователь вводит данные карты и проводит оплату на стороне провайдера.
5. **Метод СБП:** Сервис генерирует QR-код: с помощью класса SBPIntegration формирует JSON с merchant_id, суммой, описанием и transaction_id [41] [159], кодирует его в QR (PNG картинка) и преобразует в base64 строку [160]. Возвращает объект SBPQRCode с полями: `qr_code` (base64 PNG) и `payment_url` (специальная ссылка sbp://pay?...). Payment запись обновляется: `payment_provider="sbp"`, `payment_metadata` = JSON с данными QR [47] [161]. Статус остаётся "pending". Клиент получает base64 QR, отображает его – пользователь

может сканировать через приложение своего банка для оплаты. После сканирования платеж приходит на счет клиники (или агрегатора).

6. **Метод "Кошелек":** Сервис проверяет, есть ли у пользователя кошелек и достаточно ли на балансе средств [48]. Если нет – возвращает ошибку **400** ("Недостаточно средств") [162]. Если да – сразу списывает сумму: уменьшает `wallet.balance` и помечает Payment как `completed` [48]. Фиксируется транзакция списания (в коде это реализовано внутри update кошелька, рассмотрено ниже). По сути, оплата мгновенно проходит, никаких внешних интеграций.

7. **[Завершение платежа]:**

8. В случае **YooKassa/SBP**, после того как пользователь оплатит на стороне банка, YooKassa вызовет callback (webhook) на указанный адрес (если настроено) или клиент вернется по return_url. В MVP есть эндпоинт `POST /payments/{payment_id}/complete` [163], видимо, для webhook: он просто ставит статус платежа `completed` без дополнительных проверок (предполагая, что YooKassa подтверждает, а связь Payment ID известна через metadata) [164]. Аналогично, на возврат средств предусмотрен `/payments/{id}/refund` (меняет status на refunded) [165] [166] – его можно дергать вручную, если возврат согласован.

9. Для **внутреннего кошелька** всё уже произошло моментально: Payment был помечен `completed` в момент списания. Тем не менее, можно фиксировать операцию – и в коде действительно при депозите/снятии создаются `Transaction` записи.

10. **[Внутренний кошелек и транзакции]:**

11. Создание кошелька: при регистрации кошелек не создается по умолчанию, но админ или система может вызвать `POST /wallets/` с user_id. Сервис проверит, что кошелька нет, и создаст со стартовым балансом 0 [167] [168].
12. Пополнение кошелька: `POST /wallets/{user_id}/deposit?amount=X`. Проверяет, что `current_user_id` совпадает, иначе 403 [169]. Затем увеличивает баланс на сумму [170] [171] и создает запись Transaction (deposit) [172]. Возвращает новый баланс [173].
13. Списание с кошелька: `POST /wallets/{user_id}/withdraw?amount=Y`. Аналогично: проверка владельца [174], достаточно ли средств [175], уменьшение баланса и запись Transaction (withdrawal) [176] [177], возврат успеха с новым балансом [178].
14. Просмотр транзакций: `GET /transactions/wallet/{wallet_id}` вернет все операции по конкретному кошельку [179]. `GET /transactions/user/{user_id}` – все транзакции пользователя, сначала находит кошелек, потом по нему транзакции [180] [181]. Эти методы позволяют, например, в кабинете видеть историю операций.

Оценка реализации: Для MVP платежный модуль сделан весьма основательно: интеграция с реальными платежными сервисами подготовлена, есть резервный сценарий с внутренней валютой. Однако, обнаружены и **проблемные моменты:**

- **Безопасность вызовов Payment Service:** Обратите внимание, `create_payment` **не проверяет**, что `user_id` в запросе совпадает с авторизованным пользователем. По Gateway `/payments/create` доступен любому залогиненному, Gateway передаст `X-User-ID`, но Payment Service просто берет `payment.user_id` из тела JSON [43]. Злоумышленник мог бы попытаться создать платеж от чужого имени (указав другой user_id). В идеале, `user_id` не должен передаваться вообще – сервис мог определять его по токену. Либо Gateway должен блокировать несоответствие. Сейчас такого механизма

нет, что является **уязвимостью**: требуется доверять клиенту. В продакшене это нужно исправить.

- **Несогласованность идентификаторов:** Как и в Profile Service, Payment Service использует `user_id` типа Integer для Wallet/Payment [37] [38]. Это опять же не синхронизировано с UUID пользователей. Возможно, разработчики рассчитывали на миграцию к числовым ID (например, сделать отдельный автоперечисляемый ID для каждого пользователя). Но пока этого нет – записи Payment и Wallet фактически не связаны внешним ключом с user в SQLite. Это не мешает хранить, но, например, Payment Service считает, что Payment.user_id `5` – это "пользователь с ID 5", тогда как User Service может не иметь такого (uuid совсем другое). Если система используется целостно, эти ID могут никогда не сопоставляться явно. Но если нужно отобразить платежи пользователя, идет запрос `/payments/user/{user_id}` с user_id=UUID? Однако, Payment Service ждет int и не найдет запись. Видно, что Payment Service писался под предположение, что user_id – числовой идентификатор. Этот **дизайн-изъян** следует устранить: либо хранить user_id как строку (и тогда Wallet.user_id = user.uuid), либо ввести глобальный numerical ID.

- **Обработка результатов платежа:** В MVP оплата через карту/СБП не доведена до конца: нет кода, который по webhook меняет статус заказа на "оплачено" или уведомляет врача. Хотя Payment Service меняет статус платежа на completed, это изолировано. Предположительно, админ или backend могли бы по событию оплаты изменить состояние **Booking** (например, отметить заказ как подтвержденный, или разблокировать кнопку "Начать визит"). Пока такой связи нет. Можно дополнить: Payment Service при завершении мог бы бросать событие или вызывать Booking Service (например, выставлять `Booking.status = paid` – но такого статуса нет).

- **UI и взаимодействие:** В admin-панели заявлена страница "Платежи" (баланс и история транзакций) [182], значит, часть этих эндпоинтов используется. Вероятно, admin-panel дергает `/api/payments` (который, к слову, нет в gateway), или прямые `/payments` эндпоинты. Пациентскому приложению тоже нужна страница кошелька и оплаты. Эти компоненты в MVP могут быть не полностью интегрированы, но задатки есть. QR-код СБП, например, нужно отобразить – пока Payment Service возвращает JSON с base64, frontend должен уметь декодировать его в картинку.

- **Кошелек:** Логика кошелька работает, однако нет автоматического создания кошелька при регистрации – только по запросу. Если пользователь не знает о кошельке, он и не появится. В admin-панели, возможно, можно создавать кошелек для пользователя (например, при выдаче бонусов). Также при удалении пользователя стоит удалять кошелек, но об этом не сказано (нет связи – будет висячая запись). Это мелочи для MVP, но важно для полноты.

В целом, **платежная система MVP близка к завершенной** в серверной части: есть интеграции, управление статусами, учет баланса. Требуются некоторые доработки: защита эндпоинтов, синхронизация ID, а также тестирование реальных платежных потоков. До продакшна необходимо убедиться, что оплата действительно удобна: например, внедрить **уведомление** Payment Service о результатах (webhook от YooKassa должен быть настроен, и endpoint secure), а клиент – обработать результат (успех/отмена). **Модульность** решения хороша: Payment Service почти автономен, и его можно развивать независимо (например, добавить другие методы оплаты).

6. Администрирование и модерация

Описание процесса: Администратор системы выполняет несколько задач – управление пользователями (блокировка/удаление), проверка и допуск новых врачей/клиник, мониторинг заказов, генерация отчетов. В проекте предусмотрено отдельное веб-приложение Admin Panel и набор API для него. Рассмотрим ключевые процессы модерации:

- **Управление пользователями:** Админ-панель имеет раздел “Пользователи” [183]. При открытии списка отправляется запрос `GET /api/users` (через Gateway), который проксируется на User Service `/users` [184] [185]. Gateway проверяет JWT администратора [185] и разрешает получить страницу пользователей с пагинацией. User Service возвращает список UserResponse и общие метрики (total, pages) [186] [187]. Админ видит таблицу пользователей. Он может просмотреть отдельного пользователя (`GET /api/users/{id}`) – пойдет на User Service `/users/{id}` [188], получить детали (роль, активность и т.п.) [189] [190]. Также доступны операции: **обновить пользователя** (например, сменить роль, активировать/деактивировать, пометить проверенным – `PUT /api/users/{id}`) [191], и **удалить пользователя** (`DELETE /api/users/{id}`) [192]. Gateway требует роль admin для всех этих действий [193] [194]. В User Service соответствующие методы реализованы: обновление происходит через тот же `/users/{id}` endpoint, где можно изменить поля (в коде сейчас не явно выделено, но можно переиспользовать модель и обновить нужные атрибуты), удаление – удаляет запись из БД [195] [196] (в реальном коде delete user не был явно в нашем просмотре, но упомянут в API). Таким образом, администратор может **деактивировать учетку пациента** (поставив `is_active=False`, тогда при логине будет отказ) или **повысить роль** (например, назначить какого-то пользователя админом или врачом). Процесс **модерации врачей** может выглядеть так: врач регистрируется (как patient или сразу role=doctor, is_verified=False), админ проверяет его документы офлайн, затем в админке меняет `is_verified=True` и, возможно, роль на `doctor` (если не было). Прямо отдельного эндпоинта "approve doctor" нет, делается через обновление пользователя.

- **Управление врачами и клиниками:** Существуют разделы “Врачи” и “Клиники” в админке [197]. Они, скорее всего, показывают списки профилей (плюс связанные данные из users). Запрос `GET /api/doctors` Gateway шлет на Profile Service `/doctor-profiles/` [198] [199], который возвращает список всех записей DoctorProfile (можно опционально фильтровать по специализации или доступности) [200]. Аналогично `/api/clinics` -> `/clinic-profiles/` [201] [202]. Админ видит таблицы врачей/клиник. Добавление нового врача админом: нажимает “Добавить врача”, вводит email, имя и др. Вероятный сценарий – сначала создать учетную запись (через тот же API Users: `POST /api/users`), указав роль doctor [203], а затем создать профиль врача: `POST /api/doctors` (Gateway -> `/doctor-profiles/` в Profile Service) [204] [205]. Однако, в Gateway `POST /api/users` реализован как прокси на `/auth/register` User Service [203] [206]. Это зарегистрирует нового пользователя и вернет токен, что, возможно, не нужно в контексте админ-панели (админ не будет использовать чужой токен). Вероятно, фронтенд игнорирует токен в ответе, используя лишь данные созданного пользователя. Затем `POST /api/doctors` с данными профиля (включая user_id, который админ выберет или автоматически подставит). Этот процесс не подтвержден документацией, но вытекает из предоставленных API. Обновление профиля врача (`PUT /api/doctors/{id}`) или удаление (`DELETE /api/doctors/{id}`) через Gateway идет на Profile Service [207]. Удаление врача, вероятно, только удаляет профиль, не трогая саму учетную запись (не совсем консистентно – если врача исключают, стоило бы либо деактивировать user, либо удалять оба, но пока явно не указано).

- **Модерация контента и других данных:** Также заявлены разделы “Отчеты”, “Настройки” [208]. **Отчеты** могут генерироваться на основе статистики – для этого есть эндпоинт `/api/dashboard/stats` [209], который собирает количество пользователей, врачей, клиник и общее число записей [210] [211]. Сейчас показатели активных/выполненных заказов стоят заглушки “0” с TODO-комментарием [212]. В admin-панели на дашборде эти цифры отображаются. Настройки, вероятно, статические (или изменяют .env – не реализовано).

Возможная задача модерации – **отзывы и рейтинги**: пациенты могут оставить отзыв о враче после визита. Endpoints для рейтингов существуют (`POST /ratings`, `GET /ratings/doctors/{doctor_id}`) [213], но **Rating Service** не реализован (порт 8008 пуст). Скорее всего, отзывы пока не включены в MVP, либо их хранение не завершено. Админ мог бы в будущем модерировать отзывы (удалять нецензурные), но сейчас это вне рамок MVP.

- **Безопасность admin-доступа:** Все рассмотренные `/api/...` маршруты на Gateway требуют роль admin [185] [193]. Таким образом, даже если злоумышленник получит обычный токен, он не сможет воспользоваться admin API. JWT для админа ничем, кроме поля role, не отличается; управление выдачей admin-токенов – вне описания (видимо, первого администратора заводят вручную).

Оценка реализации: Функциональность администрирования для MVP выглядит убедительно:

- Admin-панель имеет доступ ко всем нужным данным через соответствующие эндпоинты. В changelog отмечено, что добавлены все недостающие API, и проверка роли admin встроена [214] [215]. Это означает, что разработчики сопоставили возможности UI с сервером и устранили пробелы (например, добавили `/api/patients` для списка всех пациентов с role filter) [216] [217]. Кстати, `/api/patients` проксируется на User Service `/users?role=patient` [218]. В самом User Service параметр role пока не обрабатывается явно (код не фильтрует по role), но, возможно, был добавлен позже или фронт просто фильтрует полученные users.

- Модерация новых врачей: нет явного workflow (“подтвердить врача”), но с помощью комбинации операций (редактировать пользователя, создать профиль) админ решает задачу. То есть **процесс онбординга врача** требует ручных шагов в админке. Это допустимо на первых этапах, однако можно улучшить: например, позволить врачу регистрироваться и загружать документы, после чего админ в одном окне видит “Новые заявки врачей” и нажимает “Одобрить” – тогда система могла бы автоматически выставить флаг и создать профиль. Пока этого нет, админ должен знать, что делать.

- **Ошибки и исключения:** В целом, API для админа опирается на те же методы, что и для основной логики, поэтому унаследованы те же проблемы (например, история с UUID vs int тоже затрагивает админ-функции: Profile Service, Payment Service могут неправильно работать с идентификаторами). Но предполагается, что админ работает в контролируемой среде и может обойти проблемы (например, просмотреть профиль врача через /api/users, а не /api/doctors, если тот не создал профиль).

- **Отсутствие отдельных admin-прав в сервисах:** Сейчас проверка “role==admin” делается на Gateway [8], а внутри микросервисов админ не имеет особых привилегий, кроме доступа ко всем данным. Это нормальный подход (централизованная авторизация). Возможно, стоило бы и на уровне сервисов некоторые операции ограничивать (например, удалить пользователя – только админ), но Gateway уже фильтрует.

В сумме **уровень модерации MVP достаточный**: администратор обладает инструментами для поддержания системы – может управлять пользователями и данными справочников (врачи, клиники), отслеживать основные метрики, вручную влиять на заказы (назначать врачей). Выявленные неполноты не критичны на MVP-стадии, однако для масштабирования **процессы администрирования следует автоматизировать и упростить**. Например, реализовать интерфейс принятия заявок врачей, добавить подтверждение контактов (почты врачей/пациентов), расширить раздел аналитики (на основе собранных данных о заказах, платежах). Также стоит развивать **Notification/Chat Service** для облегчения работы админа (например, система могла бы сама рассылать уведомления о новых регистрациях врачей для проверки).

После рассмотрения основных процессов выделим **общие проблемы и несоответствия**, обнаруженные при аудите, а затем предложим рекомендации по улучшению.

Таблица обнаруженных проблем и несоответствий

Проблема / Антипаттерн	Описание и подтверждение	Влияние на систему
Несоответствие типов идентификаторов	В разных сервисах используются разные типы для user_id: строковый UUID в User Service vs Integer в Profile и Payment. Код Profile Service ожидает int `user_id` [22], но получает UUID, а Payment Service сохраняет user_id как int, не связанный с реальным UUID [10] [37].	Связи между сервисами на уровне данных нарушаются – профили и платежи трудно однозначно сопоставить с пользователями. В худшем случае, обращения по ID могут возвращать пустые результаты или ошибку приведения типа. Требуется унификация идентификаторов (например, перейти полностью на UUID).
JWT аутентификация непоследовательна	Gateway проверяет JWT и передает идентификатор пользователя в заголовках, однако Profile Service и Payment Service пытаются самостоятельно парсить Authorization заголовок [25] [49]. Эти заглушки не учитывают реальный механизм Gateway (использующий X-User-ID).	В текущей реализации методы обновления профиля и операций с кошельком/платежами всегда будут возвращать **401**, т.к. не получают ожидаемого формата авторизации. Пользователь не сможет обновить свой профиль через API без вмешательства. Нужно исправить получение current_user_id в этих сервисах (например, читать X-User-ID, либо полагаться на Gateway полностью).

Проблема / Антипаттерн	Описание и подтверждение	Влияние на систему
Дублирование и рассинхронизация данных	Данные о врачах хранятся сразу в двух местах: User Service (специализация, лицензия и т.д. в таблице users) [11] и Profile Service (в таблице doctor_profiles) [19]. При обновлении профиля врача можно изменить, например, `license_number` в Profile Service, но поле `users.license_number` останется старым.	Возможна путаница и устаревшие сведения: фронтенд неясно, откуда брать “истинные” данные. Вероятно, планировалось убрать дубль из User или синхронизировать при обновлении. В продакшене желательно хранить информацию в одном месте (минимум – сделать явную связь: при создании профиля перенести поля из User и очистить дубли).
Неавтоматизированное создание профилей	После регистрации пользователя не создается запись в Profile Service, хотя запросы `/profiles/{id}` ожидают её [87]. Админ должен вручную создавать профили врачей, а пациенты могут вообще не иметь профиля (если не заполнят).	Функциональность профиля пациентов может остаться неиспользованной – важная информация (адрес, полис и т.д.) не будет привязана к аккаунту, если пользователь не создаст профиль вручную. Нужно либо создавать профиль по дефолту (пусть пустой), либо интегрировать запрос на профиль в клиент (например, при первой авторизации предлагать заполнить).
Отсутствие проверки прав при создании платежа	Метод создания платежа (`/payments/create`) не сопоставляет `payment.user_id` с текущим пользователем [43]. Можно отправить запрос от имени А с `user_id` Б, и Payment Service его примет. Gateway тоже это не фильтрует.	Это **проблема безопасности:** злоумышленник мог бы попытаться осуществить платеж за другого пользователя или разведать информацию. На практике, для оплаты нужно владеть платежными данными, но сам факт создания “чужого” платежа неправилен. Следует получать user_id из токена (не из внешнего ввода) на стороне Payment Service либо проверять на Gateway.

Проблема / Антипаттерн	Описание и подтверждение	Влияние на систему
Многопоточное использование SQLite	Все сервисы по умолчанию нацелены на одну БД SQLite файл (`sqlite:///./tot_mvp.db` без уникальных имен баз) [51] [52]. Если запустить микросервисы параллельно локально, они будут писать в один файл.	SQLite не предназначена для одновременной записи из нескольких процессов – возможны блокировки и потеря данных. Для разработки это терпимо, но для продакшена нужен переход на СУБД серверного типа (PostgreSQL, MySQL) или разделение баз (каждый сервис со своей БД). В документации упомянуто, что для локальной разработки SQLite, но вопрос конкурентного доступа открыт.
Незавершенные интеграции (чат, гео, нотиф)	В Gateway реализованы маршруты для Geo, Chat, Rating, Event, Emergency сервисов [219], но сами эти сервисы отсутствуют или не дописаны. Например, Chat Service (порт 8007) – нет реализации, вместо него работает чат внутри Booking; Rating Service – не реализован.	Пользовательские запросы по этим маршрутам вернут **503 Service Unavailable** [220]. Фронтенд MVP, возможно, даже не вызывает их (страницы “Мероприятия”, “Оценки” могут быть заглушками). Однако, наличие неработающих эндпоинтов может сбивать с толку разработчиков. Рекомендуется либо скрыть их из Gateway до реализации, либо пометить “в разработке”, чтобы не считать это багом у пользователей.

Проблема / Антипаттерн	Описание и подтверждение	Влияние на систему
Минорные логические упущения	1) В User Service timestamps создаются через `datetime.now` (локальное время) [221], тогда как другие сервисы используют `datetime.utcnow` [222] – может приводить к разнице часовых поясов. 2) В Payment Service поле `payment_metadata` хранится как JSON-строка, но в Pydantic-модели PaymentResponse оно объявлено как dict [223] [224]. При возврате ответа поле `metadata` может оказаться `null`, т.к. `payment_metadata` не маппится автоматически на `metadata`.	1) Разные таймстампы осложняют объединение логов и могут вызвать неверное истечение токена (если где-то забыли исправить). 2) Клиентская часть при оплате не получит расшифровку платежа (напр., URL для оплаты) из Response, хотя сервис его сохранил. Эти мелочи легко исправить: использовать единый подход к времени (UTC везде) и поправить модель (в PaymentResponse добавить `payment_metadata: str` или парсить JSON).

(Примечание: Дополнительные проблемы, такие как отсутствие тестов, отсутствие капчи при регистрации, потенциальная нагрузка на Gateway, тут не перечислены, сосредоточены основные логические несоответствия.)

Рекомендации по улучшению (рефакторинг и новые практики)

На основе обнаруженных недостатков предлагается ряд мер для повышения надежности, читабельности и масштабируемости системы:

- **Унификация идентификаторов и связей:** Следует перейти к единому формату ID пользователей во всех сервисах. Наиболее просто – хранить UUID как строку в Profile и Payment сервисах (поменять тип колонок user_id на String). Либо, если нужен числовой ID, использовать автоинкремент для users и сохранять этот int как внешний ключ в профилях/платежах. Главное – обеспечить уникальность и взаимосвязь. После выбора стратегии – обновить логику Gateway и сервисов, чтобы user_id не передавался в теле там, где его можно взять из JWT (исключить дублирование в запросах).

- **Доработка межсервисной авторизации:** Внедрить централизованный **Auth/Security Service** или по крайней мере общий модуль для проверки JWT в микросервисах. Сейчас Gateway проверяет токен, но доверие на уровне сервисов минимально. Можно усилить:

- Передавать в `X-User-ID` не просто raw ID, а, например, JWT-сигнатуру или отдельный service-token.
- Или дать сервисам уметь проверять подпись JWT самостоятельно (распространив секрет).
- Менее затратно: исправить Profile/Payment сервисы на использование `X-User-ID` – то есть функция `get_user_from_header` должна вытаскивать ID из `X-User-ID` (и,

возможно, роль из `X-User-Role` для контроля доступа). Это быстро решит проблему 401 при обновлении профиля и работе с кошельком.

- На будущее: **Security Service** (порт 8011 зарезервирован) мог бы заниматься выдачей/проверкой токенов, тогда микросервисы будут обращаться к нему. В рамках MVP достаточно корректно прокидывать идентичность пользователя.

- **Разделение ответственности User vs Profile:** Решить, какие данные хранятся в User Service, а какие в Profile. Рекомендуется:

- Оставить в **User** только аутентификационную и базовую информацию (имя, email, роль, активность). Убрать из User поля специализации, лицензий, адресов – это все относится к профилям.
- Все медицинские и профильные данные хранить в **Profile Service**. Для удобства, можно при регистрации врача сразу создавать DoctorProfile (заполнив известные поля). Аналогично, при регистрации клиники – ClinicProfile.
- Удалить дублирующиеся поля из модели User или хотя бы не использовать их. Сейчас при получении пользователя через `/auth/me` возвращаются, например, `specialization` [13], которые могут не обновляться. Лучше выдавать обобщенную информацию, а детали по отдельному запросу профиля.

- Таким образом, **Single Source of Truth**: профиль хранится в одном месте. Это устранит потенциальную рассинхронизацию.

- **Интеграция создания профиля и кошелька:** Автоматизировать создание связанных сущностей:

- После успешной регистрации (`/auth/register`), можно внутри User Service сделать вызов к Profile Service для создания пустого профиля (в фоновом потоке или синхронно). Либо Gateway, получив ответ от `/auth/register`, сам дергает `/profiles/` с необходимым user_id. Это избавит от ситуации “Profile not found” при первом обращении.
- По аналогии, сразу создавать **Wallet** для нового пользователя (если концепция кошелька предполагается для всех). Или хотя бы для врача, если врачам нужен депозит для взаимодействия.

- Если по бизнес-логике не всем нужен кошелек, можно отложить до первого платежа – например, при попытке списания, если кошелька нет, создать автоматически. В Payment Service можно вставить логику: при `/wallets/withdraw` если wallet не найден – создать и тут же выдать ошибку “недостаточно средств” (так кошелек появится).

- **Усовершенствование процесса назначения врача:** Чтобы разгрузить админа, стоит реализовать **алгоритм подбора врача**. Это крупное улучшение, но даже небольшой шаг – например, при создании заказа сразу искать свободных врачей указанной специализации через Profile Service и отправлять им уведомление (Telegram-бот или push). FSM (автомат состояний) здесь можно применить: модель заказа уже фактически FSM (pending -> assigned -> in_progress -> completed -> cancelled), можно формально описать эти состояния и переходы, чтобы гарантировать корректность. Например, реализовать **паттерн State** или использовать библиотеку FSM для Python, которая предотвратит неразрешенные смены статуса (хотя они и так предотвращены условными if’ами). Для бот-алгоритма FSM: можно шаги записи (выбор специализации, времени, подтверждение оплаты) реализовать как конечный автомат диалога, что улучшит UX.

- **Подключение Telegram-бота для врачей и оповещений:** Раз пользовательские приложения для врачей пока нет, целесообразно внедрить бот. Aiogram (уже фигурировал в репо) может служить для:

  - Уведомления врача о новом назначенном визите (бот шлет сообщение с деталями заказа).
  - Врач через бот может нажать “Принять визит” (что по API вызовет `/bookings/{id}/start`), “Завершить визит” (`/complete`), или написать сообщение пациенту (бот -> backend -> сохранение BookingMessage).
  - Пациенту, в принципе, тоже можно дублировать уведомления в бот (если Telegram-ID привязать).

Для этого нужно расширить систему: хранить Telegram chat_id в профиле пользователя, и реализовать в боте StateMachine для каждого процесса (например, подтверждение назначения: бот спрашивает врача “Вы примете вызов от такого-то пациента?”, врач отвечает да – бот делает API-запрос assign/start).

Такой бот повысит оперативность связи. На backend потребуется endpoint для бота или прямые вызовы бота к API (бот может действовать как админ/врач используя JWT). Это организационные моменты, но **FSM для бота** как раз пригодится.

- **Улучшение уведомлений и реального времени:** Как альтернатива или вместе с ботом – **Notification Service**. Его зачатки видны (порт 8006), но можно проще: использовать WebSocket в Booking Service. Например, при поступлении сообщения в чат – рассылать через WebSocket/SocketIO всем клиентам заказа. Или при смене статуса заказа – пушить обновление на фронтенд. Сейчас этого нет: клиент должен опрашивать `/bookings/{id}` статус. Для MVP приемлемо, но в продакшене лучше push-модель. Notification Service мог бы обрабатывать сообщения и статусы и разносить по подписчикам (пациентскому приложению, админке, боту и т.д.).

- **Refactoring кода и организационная чистота:** Несмотря на то, что MVP можно держать сервисы в одном файле, при росте функционала стоит **разбить код на модули**:

  - Выделить модели и схемы (Pydantic) в отдельные файлы, роуты – в другие, сервисные утилиты (например, интеграция с YooKassa) – в третьи. Сейчас, например, файл Payment Service ~600 строк – можно вынести класс SBPIntegration, YooKassaIntegration в `integrations.py`, модели Payment/Wallet/Transaction – в `models.py`, а сами endpoints – в `routes.py`. Это повысит поддерживаемость.
  - Аналогично для Booking Service: endpoints CRUD vs chat можно разделить, модель StatusUpdate вынести.

Также обратить внимание на **повторяющийся код формирования ответа**. Много раз создаются объекты Response вручную (копируя поля модели) [225] [226]. Можно упростить: Pydantic позволяет использовать `response_model` и возвращать напрямую объект SQLAlchemy – он сам преобразуется (если настроен `Config.from_orm = True`, либо вызывая Pydantic модель с объектом). В коде уже отчасти это сделано (например, `return booking` при response_model=BookingResponse может работать, если Config.from_attributes, но где-то писали явно).

Кроме того, **обработка ошибок**: сейчас логика if/raise повсюду. Возможно, для повторяющихся проверок (например, “пользователь не найден”, “доступ запрещен”) стоит сделать небольшие функции-хелперы, чтобы сократить код и избежать потенциальных опечаток в сообщениях.

- **Переход на production-ready технологии:** Это перспектива, но указать нужно:
- Перенести хранение данных с SQLite на PostgreSQL/MySQL. Уже сейчас используется SQLAlchemy, так что смена движка несложна. Шаги: поднять инстансы БД для каждого сервиса (либо единую схему и настроить соединения). Это устранит проблемы с параллельным доступом и обеспечит масштабирование.
- Docker-изация: В репо вероятно есть `docker-compose.yml` – нужно убедиться, что все сервисы работают в контейнерах, настроить персистентность БД, подключить безопасно секреты (например, переменные для токенов и платежей).
- Логирование и мониторинг: Включить более подробные логи действий (кто назначил врача, ID платежа при ошибках и т.д.) [53] . Интегрировать maybe Sentry или аналог для отслеживания ошибок runtime.

- Тестирование: написать unit-тесты на критичные функции (регистрация, логин, создание заказа, смена статуса, оплата). В changelog упоминается `scripts/test-api-endpoints.ps1` [227] – возможно, это набор интеграционных тестов PowerShell. Стоит развить автоматическое тестирование (CI pipeline), прежде чем выпускать обновления.

- **Расширение бизнес-логики:** Некоторые запланированные функции можно постепенно реализовать:

- **Рейтинги врачей:** После завершенного визита позволять пациенту отправить оценку и отзыв. Для этого дописать Rating Service: сохранять отзыв, обновлять средний рейтинг врача (поле `rating` и `total_reviews` уже есть в DoctorProfile [228]). Админ мог бы модерировать отзывы (удалять неприемлемые).
- **Экстренные вызовы:** Имеются маршруты `/emergency/alert` и `/emergency/{id}/status` [229] , можно довести до ума: это, вероятно, отдельный сценарий, когда пациент жмет “SOS” – тогда админ/система должна мгновенно найти ближайшего свободного врача или вызвать скорую. Пока эти эндпоинты заглушки, но идея важная. Требует интеграции с Push/ботом для моментального реагирования.
- **Мероприятия:** Возможно, подразумеваются какие-то акции или выездные сессии – сейчас сервис Events не реализован. Можно либо убрать из MVP, либо спланировать – но это необязательная фича.
- **Разграничение ролей врача и клиники:** В коде роли doctor и clinic присутствуют, но **приложение для клиники** (Clinic Web) только планируется [230] . Нужно будет учесть, что клиника может иметь несколько врачей, и процессы (например, клиника назначает своего сотрудника на вызов). Пока же role=clinic не задействована. На будущее – разработать отдельные кейсы (возможно, в админке частично закрыты).

Практики, которые можно внедрить:

- **Использование FSM там, где уместно:** Мы уже обсудили FSM для чат-бота и для статусов заказа. Это поможет сделать код более структурированным. Например, библиотека `transitions` для Python может модель Booking превратить в класс с явными переходами, методами on_enter/on_exit. Это не обязательное требование, но улучшит поддерживаемость сложных процессов (особенно, когда добавятся промежуточные состояния типа "ожидание оплаты" или "отменено админом").

- **Granular access control:** Пока роль admin – единственный уровень привилегий. Можно предусмотреть в будущем более тонкие роли или права (например, **диспетчер** – может назначать врачей, но не видеть финансы; **супер-админ** – полный доступ и т.п.). Сейчас для MVP это избыточно, но архитектура должна позволять: токены JWT уже содержат `role`, можно расширить значениями или вводить клеймы (scopes).

- **API версионирование и документация:** На этапе MVP, API в одном варианте. Но стоит подумать о версионировании (добавить префикс `/v1/`) и оформить Swagger-документацию. FastAPI генерирует docs автоматически (достаточно посмотреть `/docs`), описания уже проставлены к методам 231 232 . Для внешних интеграций (например, мобильное приложение Flutter) пригодится стабильный API контракт.

- **UI/UX улучшения за счет backend:** Некоторые вещи можно упростить для фронта. Например, сейчас, чтобы админ добавил врача, нужно дернуть два эндпоинта (создать user, затем профиль). Можно сделать единый `/api/doctors` POST, который внутри Gateway вызовет оба сервиса. И фронтенду проще, и консистентней. Такие композиционные операции Gateway уже делает для статистики (агрегирует ответы нескольких сервисов) 210 211 . Можно расширить эту практику.

- **Очистка неиспользуемого кода:** Удалить или пометить `@todo` для всего, что временно не реализовано, чтобы разработчики не запутались. Например, упомянутый Payment.metadata – стоит привести в порядок модель. Или убрать из User Service устаревший метод verify_token (Gateway уже проверяет токен). Поддерживая чистоту, команда сможет быстрее разбираться в коде.

Финальное заключение

Проект **ТОТ – Твоя Точка Опоры** в текущем виде представляет собой **внушительное MVP**, покрывающее основные сценарии сервиса вызова врача на дом. Архитектура разделена на микросервисы, что для MVP редкость – но это сразу закладывает фундамент для масштабирования. Логика ключевых бизнес-процессов (регистрация, запись на прием, исполнение визита, оплата, админ-управление) реализована достаточно полно, хотя и с некоторыми упрощениями.

Пригодность к продакшену: С точки зрения функциональности, MVP уже позволяет осуществлять основной цикл услуги. Однако, для реального продакшена требуется устранить найденные несоответствия и довести до ума сопутствующие компоненты: - Решить технические баги (несовместимость идентификаторов, мелкие ошибки авторизации) – без этого возможны сбои и путаница в данных. - Обеспечить интерфейсную поддержку для всех ролей: без приложения/бота для врачей процесс будет односторонним. Продакшен подразумевает, что врачи активно используют систему. - Улучшить безопасность (проверки прав, защита данных, использование надежной БД). - Провести нагрузочное тестирование Gateway и сервисов – убедиться, что разбиение на микросервисы оправдано и не приводит к лишним задержкам.

Уровень зрелости архитектуры: На данный момент архитектуру можно оценить как **MVP-уровень, близкий к среднему**: - Плюсы: правильное выделение доменных зон, модульность, использование современных технологий (FastAPI, Async, ORM), уже реализованы интеграции (оплата) и заделы (бот, гео). - Минусы: некоторые решения чересчур упрощены или не доведены (авторизация внутри, дублирование данных), что характерно для прототипа. Нет явного единообразия в стилях кода между сервисами, есть временные решения.

Тем не менее, обнаруженные недостатки **не являются концептуально неразрешимыми**. Их можно устранить в рамках перехода от MVP к Beta-версии. Архитектура в целом правильно выбрана под задачи (микросервисы здесь оправданы интеграцией с разными внешними системами: платежи, уведомления и т.д., а также возможностью масштабировать отдельно нагрузку на запись, чат и прочее). Нужно лишь ее “отшлифовать” и закрыть пробелы.

Рекомендации Roadmap для MVP -> Prod:

1. **Исправление багов и консистентности данных** (ID, профили, JWT) – срочно, перед расширением функционала.
2. **Внедрение канала для врачей**: либо выпустить мобильное приложение доктора, либо временно задействовать Telegram-бот – это критично для работы сервиса.
3. **Сменить СУБД и настроить инфраструктуру**: PostgreSQL, Docker/Kubernetes деплой, конфигурация CI/CD.
4. **Завершить основные незакрытые функции**: отзывы/рейтинги, уведомления, геолокация врачей (если позиционируется как плюс сервиса – “найти ближайшего”).
5. **Рефакторинг кода и документации**: привести код к единому стилю, написать README для разработчиков (как добавлять новый сервис, например).
6. **Тестирование и безопасность**: добавить авто-тесты, проверить систему на XSS/SQL injections (FastAPI/Pydantic уже многое закрывают), настроить CORS и TrustedHost (они уже есть, но убедиться, что корректно) [233] [234] .
7. **Масштабирование при росте нагрузки**: мониторить Gateway – возможно, внедрить API Gateway на базе Nginx или Traefik перед FastAPI (для продакшена), настроить балансировщики для сервисов.

В заключение, **проект ТОТ как MVP жизнеспособен** и демонстрирует продуманный подход к архитектуре, соответствующий современным требованиям. Уровень зрелости пока **средний** – это прототип, требующий шлифовки, но не монолитный “спагетти”, а уже разложенный по компонентам каркас. Приведение системы в соответствие с перечисленными улучшениями повысит ее надежность и подготовит к выпуску в продакшн. Было уделено внимание масштабируемости и расширяемости (добавление новых сервисов не нарушит существующие – например, легко внедрить Security Service или заменить SQLite на PostgreSQL без переписывания бизнес-кода). Это хорошее свидетельство архитектурной гибкости.

Таким образом, при условии реализации рекомендованных доработок, проект **ТОТ** станет готовым для пилотного запуска: архитектура будет зрелой, а логика – целостной и соответствующей намерениям продукта. [235] [210]

---

1 2 3 4 9 16 36 56 57 135 152 182 183 197 208 230 235 README.md
https://github.com/ncux-ad/TOT_MVP/blob/6e228b03d1a32d61f60c812a6acb00a7540004ab/README.md

5 6 7 8 53 54 55 72 73 85 86 105 136 137 138 139 184 185 188 191 192 193 194 198 199 201 202 203 204 205 206 207 209 210 211 212 213 216 217 218 219 220 229 233 234 main.py
https://github.com/ncux-ad/TOT_MVP/blob/6e228b03d1a32d61f60c812a6acb00a7540004ab/backend/api-gateway/main.py

10 11 12 13 14 15 51 58 59 60 61 62 63 64 67 68 69 70 71 74 186 187 189 190 195 196 221 225 231 main.py
https://github.com/ncux-ad/TOT_MVP/blob/6e228b03d1a32d61f60c812a6acb00a7540004ab/backend/user-service/main.py

17 18 19 20 21 22 23 24 25 52 75 76 77 78 79 80 81 82 83 84 87 200 228 main.py
https://github.com/ncux-ad/TOT_MVP/blob/6e228b03d1a32d61f60c812a6acb00a7540004ab/backend/profile-service/main.py

26 27 28 29 30 31 32 33 34 35 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 140 141 142 143 144 145 146 147 148 149 150 151 153 222 226 232 main.py
https://github.com/ncux-ad/TOT_MVP/blob/6e228b03d1a32d61f60c812a6acb00a7540004ab/backend/booking-service/main.py

37 38 39 40 41 42 43 44 45 46 47 48 49 50 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 223 224 main.py
https://github.com/ncux-ad/TOT_MVP/blob/6e228b03d1a32d61f60c812a6acb00a7540004ab/backend/payment-service/main.py

65 66 214 215 227 changelog.md
https://github.com/ncux-ad/TOT_MVP/blob/6e228b03d1a32d61f60c812a6acb00a7540004ab/docs/changelog.md