ChatGPT

План архитектуры платформы Codex для SMM-маркетплейса

Описание: Представлена архитектура backend-платформы **Codex** – маркетплейса услуг SMM (накрутка показателей в YouTube и других соцсетях). Архитектура разработана с упором на масштабируемость, модульность (микросервисы) и безопасность. Платформа автоматизирована: пользователи самостоятельно регистрируются, пополняют баланс, создают заказы и следят за их выполнением в реальном времени, а система через интеграции API передает заказы сторонним провайдерам и получает обновления статусов автоматически [1]. Ниже описаны основные компоненты системы и взаимодействия между ними.

1. Общая архитектура сервисов и их взаимодействия

Платформа разделена на ряд специализированных сервисов (микросервисов), каждый из которых отвечает за свою часть функциональности. Взаимодействие происходит через HTTP/REST запросы между сервисами (синхронно) и через очереди сообщений (асинхронно) для фоновых задач. Такой подход обеспечивает слабую связанность компонентов и высокую отказоустойчивость.

Основные компоненты архитектуры и их взаимодействие: - **Внешние интерфейсы:** клиенты взаимодействуют с backend через набор REST API (публичный сайт, личный кабинет пользователя, API для реселлеров и админ-панель). Можно использовать единый точку входа/API Gateway, который маршрутизирует запросы к нужным сервисам, выполняет аутентификацию и rate-limit (защиту от перегрузки) [2]. - **Микросервисы доменного уровня:** запросы, поступающие через API, перенаправляются к соответствующим сервисам: сервис заказов, биллинга, уведомлений и др. Сервисы общаются друг с другом по внутренним API или через сообщения в очередях. Например, при создании заказа сервис заказов может вызвать сервис биллинга для проверки и резервирования средств, а затем обратиться к слою интеграции провайдеров для исполнения заказа. - **База данных:** используется реляционная СУБД (PostgreSQL) для надежного хранения данных пользователей, заказов, транзакций и пр. ACID-транзакции обеспечивают целостность финансовых операций (например, списание средств и создание заказа происходят атомарно). - **Кэш и очереди:** Redis применяется для кэширования часто запрашиваемых данных и хранения очередей задач. Очереди используются для асинхронной обработки операций (оформление заказов, обновление статусов, рассылка уведомлений), что разгружает основные потоковые запросы. В результате такая автоматизированная многопоточная архитектура позволяет обрабатывать тысячи заказов одновременно при минимальном участии человека [3].

Диаграмма потоков (условно): **Клиент** → (HTTP запрос) → **API Gateway/Backend** → **Сервисы** (Order, Billing, Provider...) → **Внешние провайдеры/API**. Обратные потоки: **Внешние провайдеры** (статусы) → **Provider Adapter** → **Order Service** → **Notification/Webhook** → **Клиент** (уведомление о выполнении). Все компоненты логически разделены и могут разворачиваться независимо, что облегчает масштабирование и обновление системы.

2. REST API

Backend предоставляет несколько наборов REST API для различных категорий пользователей. Каждый набор имеет свой набор эндпойнтов и требует соответствующей авторизации:

- **Публичный API сайта:** Доступен без аутентификации для вывода информации и начального взаимодействия.
- *Каталог услуг:* эндпойнты для получения списка доступных SMM-услуг, их описаний, тарифов и минимальных/максимальных лимитов. Эти данные кэшируются (например, в Redis) для быстрой загрузки каталога на лендинге.
- *Регистрация и вход:* методы для создания нового аккаунта пользователя, подтверждения email (если требуется), аутентификации (получение токена сессии или JWT). Пароли хранятся в захешированном виде для безопасности.
- *Пополнение баланса:* эндпойнты для создания запроса на пополнение (выбор метода оплаты: криптовалюта, банковская карта и т.д.). Может возвращаться, например, платежная ссылка или адрес кошелька. После подтверждения оплаты (callback от платежной системы) баланс пользователя пополняется.

- *Публичные страницы:* возможно, эндпойнты для FAQ, контактов поддержки и прочих публичных данных сайта.

- **API личного кабинета пользователя:** Требует авторизации (например, по JWT-токену в заголовке). Предназначен для зарегистрированных клиентов, позволяя им управлять своими заказами и финансами:

- *Управление заказами:* создание нового заказа (указываются сервис, количество, ссылка/ID целевого ресурса и др. параметры), получение статуса заказа, просмотр списка своих заказов (история) с фильтрацией/пагинацией. Новый заказ передается в Order Service, который сразу возвращает подтверждение создания заказа и его внутренний ID. Статусы заказов обновляются автоматически и доступны через API [4].
- *Баланс и транзакции:* получение текущего баланса кошелька, просмотр истории транзакций (пополнения, списания за заказы, возвраты). Эти данные приходят из Billing/Wallet сервиса.
- *Управление API-ключами:* если пользователь выступает как реселлер или хочет автоматизировать заказы, в кабинете он может сгенерировать API-ключ(и). Эндпойнты для создания нового ключа, отзыва старого, просмотр существующих ключей и ограничений (например, привязка к IP). Генерированный ключ показывается только один раз при создании (после чего хранится только его хеш в базе) [5].

- *Профиль и настройки:* просмотр и редактирование информации аккаунта (имя, контактные данные), настройки уведомлений (например, пользователь может включить email-уведомление о выполнении заказа).

- **API для сотрудников/менеджеров (админ-панель):** Доступен только пользователям с ролевыми правами оператора/администратора (RBAC). Могут быть реализованы на отдельном subdomain или under `/admin` и требуют аутентификации с повышенными правами.

- *Управление заказами:* просмотр всех заказов системы, фильтрация по пользователям, статусам, услугам; принудительное обновление статуса или отмена заказа; переотправка заказа провайдеру; распределение заказов между провайдерами (в случае мануального выбора); разрешение спорных ситуаций (например, начисление частичного возврата).
- *Управление пользователями:* поиск пользователей, просмотр профиля, статистики; изменение баланса пользователя (например, вручную зачислить бонус или откорректировать ошибку); блокировка/разблокировка аккаунтов при нарушениях.
- *Управление услугами и провайдерами:* добавление/удаление SMM-услуг в каталог (с указанием связанного провайдера и себестоимости), обновление цен и описаний; подключение новых провайдеров (настройка их API URL, ключей, списка предоставляемых услуг); мониторинг состояния провайдеров (возможность временно отключить провайдера, если с ним проблемы).
- *Финансы и отчеты:* обзор суммарных показателей – оборот средств, прибыль (разница между ценой продажи и себестоимостью от провайдера), количество заказов за период; просмотр журнала транзакций (депозиты, списания, рефанды) для аудита.
- *Система и настройки:* управление внутренними настройками платформы (например, параметры rate limit, ключи интеграции платежных шлюзов, шаблоны уведомлений, промо-коды и акции и т.п.).

Каждый из API-наборов может быть версионирован (например, `/api/v1/`) для обеспечения обратной совместимости при развитии системы. **Формат данных** – как правило JSON. Для защиты от CSRF и сжатия трафика используется только HTTPS. Документация по всем публичным/партнерским методам будет предоставлена, включая примеры запросов и ответов.

3. Микросервисы и модули платформы

Архитектура разделена на несколько основных сервисов/модулей, каждый из которых реализует свою бизнес-логику. Ниже перечислены ключевые микросервисы и их обязанности:

Order Service (сервис обработки заказов)

Назначение: управление жизненным циклом заказов SMM-услуг от создания до завершения.
Функции: - Принимает новые заказы от пользовательского API или API реселлеров. Валидирует параметры заказа (правильность ссылки/ID контента, допустимость количества в рамках минимума/максимума услуги, наличие достаточного баланса у пользователя и т.п.). - Инициирует списание или *холд* средств за заказ через Billing Service (например, помечает сумму как зарезервированную до выполнения заказа). Если баланс недостаточен, возвращает ошибку пользователю. - Передает заказ на исполнение внешнему провайдеру через **Provider Adapter Layer**. Order Service формирует унифицированный API-запрос с данными заказа (ID услуги у провайдера, количество, ссылка и доп.параметры) и отправляет его в адаптер провайдера. В ответ получает либо подтверждение и внешний ID заказа у провайдера, либо ошибку. В случае ошибки – освобождает зарезервированные средства/возвращает их на баланс, устанавливает заказу статус "ошибка". - Отслеживает выполнение заказа: сервис поддерживает статусы заказа (новый, в обработке, выполнен, частично выполнен, отменен и т.д.). Обновление статуса может происходить путем: - опроса API провайдера (через задание в очереди, например, каждые N минут запрашивать статус по внешнему ID заказа), - либо обработки webhooks от провайдера (если провайдер присылает уведомления о завершении). В обоих случаях Order Service принимает новые статусы и обновляет запись заказа в БД. -

Обрабатывает завершение заказа: при получении статуса "выполнен" или "частично выполнен/отменен" выполняет финальные действия – фиксирует итоговый результат (например, доставлено 950 из 1000 единиц), рассчитывает, требует ли возврат средств (в случае частичного невыполнения), и отправляет соответствующее событие в Billing (для возврата неиспользованных средств, если нужно) и в Notification Service (для уведомления пользователя о завершении). - Позволяет отменять заказ (если он еще не начал выполняться) – по запросу пользователя или администратора: в этом случае, если возможно отменить на стороне провайдера (через API вызов отмены), отменяет и возвращает средства пользователю. - Логирует все действия и изменения статусов для возможности аудита и отладки (возможно, в дополнительную таблицу истории статусов).

Order Service является ядром системы, обеспечивая оперативную обработку заказов. Благодаря очередям и автоматизации, заказы могут ставиться в обработку мгновенно и массово, без ручного вмешательства [6].

Billing/Wallet (биллинг и кошельки пользователей)

Назначение: учет и движение средств пользователей на платформе, интеграция с платежными системами.
Функции: - Ведет **кошелек пользователя** – виртуальный счет, с которого списываются средства за заказы и на который поступают пополнения. Реализуется таблицей `wallets` (или полем баланса в таблице пользователей) и связанными транзакциями. - Отвечает за **пополнение баланса:** интегрируется с внешними платежными шлюзами (например, криптопроцессоры для приема криптовалют, API платежных агрегаторов для карт/банковских платежей). При инициировании пополнения Billing Service генерирует запись о транзакции с состоянием "ожидание", и либо перенаправляет пользователя на оплату (для банковских методов), либо ожидает входящий перевод (для крипто). По подтверждению оплаты (callback от шлюза или мониторинг блокчейна), сервис зачисляет сумму на кошелек пользователя и фиксирует транзакцию как успешную. Платформа может поддерживать несколько методов оплаты (банковские карты, электронные деньги, криптовалюты и др.), что соответствует практике SMM-панелей [7]. - **Списание средств за заказ:** по запросу Order Service выполняет резервирование и затем списание средств. Как правило, при создании заказа сумма заказа сразу вычитается с баланса пользователя (или переводится в состояние *hold*). Если заказ впоследствии выполнен успешно, транзакция помечается завершенной; если заказ отменен или выполнен не полностью, Billing начисляет пользователю возврат средств (частично или полностью) – создается транзакция типа "refund". - **Учет транзакций (ledger):** Каждое движение средств фиксируется в таблице `ledger` (или `transactions`) с указанием типа операции (депозит, списание за заказ, возврат), суммы, связанного объекта (ID заказа или платежа) и времени. Ведется сквозная нумерация транзакций для аудита. Использование отдельной таблицы-леджера повышает надежность финансового учета: можно всегда просуммировать движения и сверить с текущим балансом [8]. - **Баланс и ограничения:** Billing Service реализует логику проверок: не позволит уйти в отрицательный баланс, может устанавливать ограничения на минимальный депозит или максимальный расход в день (для противодействия злоупотреблениям). - **Интеграция с Order Service:** работает в тесной связке – транзакция списания за заказ и создание записи заказа должны быть согласованными. В случае единой базы данных используется транзакция БД, охватывающая вставку заказа и списание в ledger. Если Billing и Order разнесены по разным сервисам/БД, потребуется шаблон саги для обеспечения целостности, но при нагрузке в тысячи заказов в день допустимо и централизованное хранение для упрощения ACID-операций. - **Роли и тарифы:** Billing может учитывать разные ценовые тарифы для разных ролей (например, у реселлеров оптовые

скидки). Но чаще ценовая политика заложена в прайс-лист услуг (реселлеры просто получают более дешевую услугу от провайдера и накручивают цену для своих клиентов).

Provider Adapter Layer (слой интеграции провайдеров)

Назначение: абстрагирует работу с множеством внешних поставщиков SMM-услуг, приводя их различные API к единому интерфейсу для Order Service.
Функции: - **Каталог провайдеров:** хранит информацию о подключенных SMM-провайдерах (таблица `providers` в БД): имя, базовый URL API, ключ/токен доступа, список услуг, которые они поддерживают, и возможные различия (например, название параметров). Провайдеры могут быть крупными сервисами (SMM-панели или боты), к которым наша платформа выступает в роли реселлера [9] [10]. - **Единый интерфейс для заказа:** предоставляет методы вроде `placeOrder(providerId, serviceCode, link, quantity, params)` для отправки заказа конкретному провайдеру. Внутри адаптер знает, как формировать запрос для каждого провайдера (например, одни требуют POST JSON, другие – формы, разные названия полей). Например, для одного провайдера нужно вызвать `POST /add` с параметрами `key, service, link, quantity` и получить JSON с order_id [11], для другого – иной эндпойнт. Adapter Layer берёт эти детали на себя. - **Обработка результатов:** парсит ответ от провайдера (успех или ошибка). В случае успеха возвращает в Order Service идентификатор заказа у провайдера (для последующего мониторинга). В случае ошибки – преобразует/логирует ошибку, может реализовывать политику повторных попыток (retry) если сбой сетевой. - **Статус заказа:** предоставляет метод `checkStatus(providerId, externalOrderId)` для получения статуса заказа у провайдера. Также может быть метод отмены `cancelOrder` и др., если провайдеры поддерживают (например, некоторым провайдерам можно отправить запрос на отмену заказа). - **Выбор провайдера:** если один и тот же тип услуги доступен от нескольких провайдеров (например, для балансировки нагрузки или разницы в цене), слой адаптера может включать логику выбора – например, отправлять на самого дешевого надежного провайдера или распределять заказы по разным API. На начальном этапе, однако, можно закреплять каждую услугу за конкретным провайдером (в таблице услуг хранить `provider_id` и `external_service_id`). - **Масштабирование провайдеров:** можно динамически включать/выключать провайдера (через настройку в БД или админке) – например, если провайдер уходит на техобслуживание, адаптер будет знать, что временно нельзя к нему обращаться.

Таким образом, Provider Adapter Layer изолирует всю "грязную работу" по интеграции с внешними системами, позволяя внутренним модулям (заказы) не зависеть от конкретных API поставщиков. Это облегчает добавление новых провайдеров или замену существующих без изменений в ядре системы.

Notification/Webhook Engine (уведомления и вебхуки)

Назначение: отвечает за отправку уведомлений пользователям и интегрированным системам, а также обработку вебхуков. Обеспечивает обратную связь и интеграцию в реальном времени.
Функции: - **Пользовательские уведомления:** рассылка сообщений о важных событиях. Примеры: письмо на email при успешном пополнении баланса или завершении заказа; всплывающее уведомление в личном кабинете; отправка SMS или сообщений в мессенджеры (если предусмотрено). Notification Engine получает события (например, от Order Service о завершении заказа) и асинхронно обрабатывает их, формируя уведомление нужного типа. Шаблоны сообщений и каналы могут настраиваться (например, email через SMTP или через сервис вроде SendGrid). -

Вебхуки для реселлеров: внешние клиенты (реселлеры) могут указать в настройках API callback-URL для получения уведомлений о статусе их заказов. В таком случае, когда заказ пользователя-реселлера выполнен или изменился статус, Notification Engine делает HTTP POST запрос на URL клиента с деталями события (например, JSON с ID заказа и новым статусом). Реализована система ретраев: если внешний вебхук не отвечает, повторять несколько попыток с увеличивающимся интервалом, чтобы гарантировать доставку. Также возможна запись логов доставки вебхуков (в таблицу, например, `webhook_logs`). - **Внутренние оповещения:** возможно, используется для уведомлений менеджеров/операторов – например, алерты о сбоях: если провайдер недоступен, или большая очередь необработанных заказов. Это можно интегрировать с внутренними каналами (Slack, Telegram-бот для админов и пр.). - **Обработка входящих webhook-ов:** помимо отправки, модуль может принимать входящие webhook-и от сторонних сервисов. В частности, платежные шлюзы (для пополнения баланса) или некоторые SMM-провайдеры могут слать уведомление, когда заказ выполнен. Notification Engine (совместно с Provider Adapter) может выступать точкой входа для таких запросов: валидировать источник (например, по секрету), парсить данные и передавать соответствующим сервисам (например, Billing или Order Service) для обработки. Таким образом обеспечивается реакция на внешние события в реальном времени. - **Очередь и масштабирование:** все отправки выполняются асинхронно через очередь заданий, чтобы не блокировать основную логику. Можно сгруппировать рассылки (batch), например, не слать сотню писем раздельно, а обработать за раз. Сервис масштабируется горизонтально при увеличении количества уведомлений.

Итог: Notification/Webhook Engine повышает качество сервиса за счет своевременных информирования клиентов и интеграции с их системами. Пользователь видит актуальный статус без постоянного ручного опроса, а реселлеры могут автоматизировать работу, получая callbacks от нашей платформы вместо постоянного polling.

Auth/RBAC (аутентификация и права доступа)

Назначение: централизованная служба аутентификации пользователей и управления ролями (RBAC - role-based access control). Обеспечивает безопасный вход в систему и разграничение доступа.
Функции: - **Регистрация и вход:** создаёт учетные записи пользователей (таблица `users` с полями id, email, хеш пароля, роль и др.). Пароли хранятся в виде стойких хешей с солью. При входе проверяется пароль, и выдается токен сессии (например, JWT), содержащий идентификатор пользователя и его роль/права. Также может поддерживаться OAuth2 или SSO при необходимости, но для собственной платформы достаточно JWT. - **Управление сессиями:** в случае JWT – сессии хранятся на стороне клиента (stateless), опционально можно хранить токены или их blacklist в Redis для возможности принудительного логаута. Если используются серверные сессии, храним идентификатор сессии-> пользователь в Redis (как in-memory store) для быстрого доступа. - **Роли и разрешения:** определены роли, например: **пользователь** (клиент, может создавать заказы для себя), **реселлер** (тот же пользователь, но использует API для заказов других клиентов – фактически отличается только тем, что у него активирован API-ключ и, возможно, специальный ценовой план), **оператор/менеджер** (сотрудник поддержки, может просматривать заказы и управлять ими), **администратор** (полные права на систему). Auth Service включает таблицу или конфигурацию разрешений для каждой роли (какие API эндпойнты и действия им разрешены). Все запросы, проходящие через API Gateway или напрямую к сервисам, проверяют токен и права. Например, обычный пользователь не может получить данные чужого заказа или доступ к админ-функциям – сервис вернет 403 Forbidden. - **Управление API-ключами:** хотя API-ключи выдает пользователь через личный кабинет, их проверка может быть задачей Auth-сервиса. Каждый запрос к API для

реселлеров содержит ключ; Auth сервис сопоставляет его с записью в `api_keys` и подставляет соответствующего пользователя (имитирует login). Ключи генерируются случайным достаточно длинным токеном (например, 32+ символов) и хранятся **в базе только в виде хеша** (одностороннее шифрование). Это означает, что оригинальный ключ не восстановить, благодаря чему даже при утечке БД злоумышленник не сможет получить ключи [12]. При создании ключ показывается один раз пользователю, далее он должен хранить его самостоятельно. - **Дополнительная безопасность:** возможно, поддержка 2FA для входа администраторов, контроль одновременных сессий, сложность паролей и т.п. Также, если нужно, интеграция с внешними IDM системами для корпоративных клиентов.

Auth/RBAC сервис отделяет логику безопасности от бизнес-логики. Это упрощает добавление новых методов аутентификации (например, социальный логин), масштабирование входа (можно вынести на отдельный сервис авторизации) и централизованное применение политик безопасности.

4. Базы данных и хранилища

Основное хранилище данных – реляционная база PostgreSQL. Выбор обусловлен надежностью, поддержкой сложных запросов и транзакций, необходимостью хранить связные данные (пользователи, заказы, платежи) с целостностью. Основные таблицы БД могут выглядеть следующим образом:

Таблица	Назначение и содержимое (колонки)
users	Учетные записи пользователей системы: `id` (PK), `email`, `password_hash`, `role` (роль/уровень доступа), контактные данные (например, `name`, `telegram`), статус (активен/заблокирован), дата регистрации и пр.
api_keys	API-ключи для доступа реселлеров: `id` (PK), `user_id` (владелец, FK -> users), `key_hash` (хеш самого ключа), возможно `key_prefix` для удобства, `allowed_ips` (опционально список/IP-маска, с которых разрешен доступ по этому ключу), `created_at`, `last_used_at` и статус (активен/отозван).
services	Справочник предлагаемых услуг SMM: `id` (PK), `name` (название, например "YouTube Subscribers"), `category` (тип/категория, напр. "YouTube"), `description`, `min_quantity`, `max_quantity`, `price_per_unit` (наша цена для клиента), `provider_id` (связаный провайдер, FK -> providers), `provider_service_id` (идентификатор услуги у провайдера), `cost_per_unit` (себестоимость, для расчета маржи), флаги/настройки (поддерживает ли частичное выполнение, refill и т.д.).
providers	Список внешних провайдеров (поставщиков услуг): `id` (PK), `name` (имя, напр. "ProviderX API"), `api_url` (базовый URL их API), `api_key` / `auth` (учетные данные для доступа), `status` (активен/выключен), `notes` (примечания, например, особенности использования, задержки). Сюда же могут храниться параметры соединения (таймауты, макс. кол-во одновременных запросов).

Таблица	Назначение и содержимое (колонки)
orders	Заказы пользователей на услуги: `id` (PK), `user_id` (чей заказ, FK -> users), `service_id` (какую услугу заказали, FK -> services), `quantity` (количество единиц, например 1000 подписчиков), `link` или `target` (ссылка на аккаунт/пост, или идентификатор цели), `status` (статус заказа: new, processing, completed, partial, cancelled, error и т.д.), `created_at`, `updated_at` (время последнего обновления статуса), `provider_id` (через какого провайдера выполняется, FK -> providers), `external_order_id` (ID заказа у провайдера, если отправлен), `cost` (списанная сумма за заказ), `remains` (сколько не выполнено, если partial), и другие технические поля (например, попытки запроса статуса, отметка о необходимости рефанда и т.п.).
wallets	Балансы пользователей (если ведутся отдельной таблицей): `user_id` (PK/FK -> users), `balance` (текущее число средств, напр. в USD или иной валюте платформы), `currency`. Однако баланс можно хранить и как поле в `users` для упрощения, а движения средств вести в отдельной таблице.
ledger	Финансовые транзакции (книга учёта движения средств): `id` (PK), `user_id` (FK -> users, чьего кошелька движение), либо `wallet_id` если отдельная таблица кошельков, `type` (тип: deposit, order_debit, refund, withdrawal), `amount` (сумма, положительная для зачислений, отрицательная для списаний или наоборот – в зависимости от соглашения), `currency`, `related_id` (optional – ссылка на связанный объект: ID заказа, ID платежа и пр.), `timestamp`, `status` (напр. pending/confirm). Эта таблица хранит полную историю операций. Например, пополнение баланса создаст запись deposit (pending), которая обновится на success, списание за заказ – запись order_debit, возврат – refund и т.д. [8].
payments *(опционально)*	Если требуется детализация пополнений: отдельная таблица для заявок на пополнение/вывод: `id`, `user_id`, `method` (например, "Visa ****1234" или "BTC"), `amount`, `status` (pending, confirmed, failed), `external_tx_id` (идентификатор транзакции у платежного провайдера), `created_at`, `confirmed_at`... Однако часто депозит можно отражать сразу в ledger без отдельной таблицы.
notifications *(опционально)*	Лог отправленных уведомлений/вебхуков: `id`, `user_id` (или `api_key_id` для webhook реселлера), `event` (тип события, напр. order_completed), `destination` (email адрес или URL вебхука), `status` (sent, failed), `attempts`, `last_attempt_at` и т.п. Используется для мониторинга и повторной отправки при сбоях.

Приведенная структура может расширяться при необходимости (например, таблица `referrals` для реферальной программы, таблица `tickets` для поддержки и пр.). При проектировании схемы важно нормализовать данные (избегать дублирования, связать ключами) и добавить нужные индексы (по полям фильтрации: user_id в orders, статусам, и т.д.) для быстрого поиска.

Хранилище Redis: используется в двух ключевых ролях: - *Кэш:* хранение часто запрашиваемых данных, чтобы не обращаться каждый раз к PostgreSQL. Например, список услуг (`services`) и цен можно кешировать на несколько минут, данные профиля пользователя, баланс (если нужно быстро показывать без постоянного расчета) и прочие справочники. Также можно кэшировать результаты тяжелых запросов админ-отчетности (с обновлением по расписанию). - *Сессии и токены:* если используются серверные сессии (например, JWT не применяются для админ-панели), Redis хранит активные сессии: быстрый доступ по session_id -> данные пользователя. При использовании JWT можно хранить в Redis список отозванных токенов (чтобы реализовать logout до истечения токена) или последние успешные логины, а также OTP-коды для 2FA. - *Очереди задач:* Redis может выступать брокером для очередей (см. следующий раздел). Например, списки задач для отправки уведомлений, либо Stream для упорядоченных событий. В небольшом масштабе это упростит архитектуру (не нужен отдельный RabbitMQ), так как многие фреймворки поддерживают фоновые задачи с Redis. - *Счетчики и rate limiting:* для реализации ограничения запросов (количество запросов в минуту) Redis удобен тем, что инкременты и проверки можно делать атомарно в распределенной среде. Например, хранить ключ `rate:user123:api_call_count` с экспирацией 1 минута. - *Locks:* в случае необходимости можно применять Redis для распределенных блокировок (например, чтобы два воркера не взяли один и тот же заказ в обработку одновременно, используя механизм SETNX).

Все критичные данные хранятся в PostgreSQL (поскольку она журналирует и обеспечивает надежность). Redis держит только временную или дублируемую информацию, потеря которой не приведет к потере бизнес-данных (в крайнем случае, просто будет перегенерирована или перезапрошена).

5. Очереди задач и асинхронная обработка

Очереди (Task Queues) играют важную роль в обработке заказов и других операций, которые не должны выполняться в основном потоке запросов. Архитектура предусматривает фоновые воркеры, обрабатывающие такие очереди. Это обеспечивает масштабируемость и устойчивость к нагрузкам, позволяя выстраивать заказы в очередь и обрабатывать их последовательно или параллельно без задержек для пользователя [3] .

Основные сценарии использования очередей: - **Очередь заказов на выполнение:** когда пользователь создает заказ, Order Service помещает задачу в очередь на отправку этого заказа внешнему провайдеру. Воркер (или пул воркеров) берёт задачу из очереди и выполняет вызов к API провайдера. Такой async-подход позволяет мгновенно реагировать пользователю (он сразу получает подтверждение, что заказ принят), а тяжелые операции происходят вне контекста HTTP-запроса. Если провайдер отвечает мгновенно, можно было бы и синхронно, но очереди дают гибкость: например, если провайдер не отвечает, задачу можно повторить, не блокируя пользователя. - **Очередь обновления статусов (polling):** для заказов в статусе "в процессе" планируется периодический опрос статуса. Можно реализовать периодические задания: например, каждые 5 минут ставить в очередь задачу "проверить статус заказа X у провайдера", и воркер вызывает Provider Adapter -> getStatus. Частоту можно варьировать в зависимости от давности заказа (чаще для новых, реже для старых). Такие задачи могут ставиться Timer-сервисом или самой Order Service (например, после отправки заказа запускать фоновые задачи по расписанию). - **Очередь уведомлений:** при возникновении события (заказ выполнен, депозит зачислен) соответствующий сервис (Order или Billing) публикует задачу на отправку уведомления. Notification Service вытаскивает ее и отправляет email или делает webhook. Если отправка не удалась (например, вебхук не ответил или почтовый

сервис вернул ошибку), задача может быть повторно помещена в очередь с пометкой попытки. Можно настроить несколько повторов с экспоненциальной задержкой. Это гарантирует, что временные сбои не приведут к потере уведомления. - **Очереди для интенсивных операций:** например, генерация отчетов в админке (чтобы не делать это синхронно) или очистка старых данных, интеграция с внешними API для обновления списка услуг от провайдера (раз в день получать актуальный перечень/цены через задачу). - **Разделение и приоритеты:** возможно создание нескольких очередей (каналов) для разного типа задач. Например, очередь `orders` (высокий приоритет), очередь `statuses` (второстепенные периодические задачи), очередь `notifications` (рассылка). Воркеры можно настраивать по группам, чтобы, скажем, задачи обновления статусов не занимали все воркеры и не задерживали более критичные задачи размещения новых заказов.

Технологическая реализация: как брокер сообщений может использоваться Redis (например, через библиотеки BullMQ для Node.js, RQ/Celery для Python, Sidekiq для Ruby и т.д.) или полноценная MQ-система (RabbitMQ, Kafka). Для умеренных нагрузок (тысячи сообщений в день) Redis прекрасно справится. Задачи хранятся в виде сериализованных данных (например, JSON с ID заказа и требуемым действием). Воркеры запускаются отдельно от веб-серверов, их число можно динамически увеличивать при росте нагрузки.

Гарантия доставки и обработка сбоев: важно настроить подтверждение выполнения задачи и повторное выполнение. Например, если воркер упал, не подтвердив задачу, она должна вернуться в очередь или быть переназначена другому воркеру. RabbitMQ и аналогичные брокеры поддерживают ack/requeue механизмы; при использовании Redis можно внедрить контроль (например, перемещать задачу из списка в временное хранилище на время выполнения, и по таймауту возвращать обратно, если воркер не завершил).

Пример потока заказа (с очередями): Пользователь создает заказ → HTTP-запрос возвращает успех (заказ принят, ID). Order Service ставит задачу "исполнить заказ ID X" в очередь. → Воркер Order-Executor получает задачу, вызывает API провайдера. Получив ответ, обновляет статус заказа (например, "запущено") и сохраняет `external_order_id`. Если сразу получен финальный статус (редко, обычно заказы выполняются какое-то время), сразу ставит задачу уведомления. Иначе ставит задачи на проверку статуса каждые N минут. → Когда провайдер через несколько итераций дает статус "completed", воркер обновляет заказ в БД, публикует событие "order_completed". → Notification воркер получает событие, рассылает уведомления (email/webhook). → Billing воркер при необходимости получает событие "refund" (если частичный возврат) и начисляет деньги. Вся эта цепочка работает асинхронно, позволяя системе масштабироваться и обрабатывать множество заказов параллельно [6] [13].

6. Масштабирование архитектуры

Проектирование выполняется с расчетом на рост нагрузки до высоких значений (тысячи заказов в день и более). Предусмотрены следующие подходы к масштабированию:

- **Горизонтальное масштабирование сервисов:** благодаря микросервисной архитектуре, каждое звено (API, Order Service, Billing, и т.д.) можно масштабировать независимо. Например, если Order Service становится узким местом, разворачиваем несколько инстансов этого

сервиса behind load balancer. Точно так же, несколько воркеров обработчиков очередей могут работать параллельно. Использование контейнеризации (Docker) и оркестрации (Kubernetes) облегчит управление масштабированием – можно динамически добавлять реплики сервисов под рост нагрузки [14]. При этом важно, чтобы сервисы были **безд状態** (stateless) – не хранили пользовательских сессий или данных локально, тогда их клонов можно запускать сколько угодно, а общие данные вынесены во внешние хранилища (БД, Redis).

- **Разделение по функционалу:** по мере роста некоторые модули можно выделять на отдельные сервера/кластеры. Например, вынести базу данных на отдельный мощный сервер или кластер с репликацией (master-slave, где master для записи, реплики для тяжелых чтений отчетности). Можно отделить сервис авторизации (Auth) в отдельное приложение, если необходимо (особенно, если будет много OAuth или SSO). Также, возможно, разделить Billing Service, если финансовые операции требуют отдельного контура безопасности.
- **Балансировка нагрузки:** для веб-уровня (REST API) ставится load balancer (например, Nginx/HAProxy или облачный аналог), который распределяет запросы между экземплярами бекенда. Этот же уровень может применять rate limiting и кэширование на уровне прокси (для некоторых GET-запросов).
- **Масштабирование БД:** PostgreSQL масштабируется вертикально (увеличением ресурсов сервера) и горизонтально через репликацию. Для высоких нагрузок можно настроить пул подключений, а также шардинг по функциональности: например, вынести "тяжелые" таблицы (ledger, логирование) в отдельную базу, чтобы нагрузки от финансового аудита не влияли на основные транзакции заказов. Но до очень больших объемов единой базы с оптимизированными индексами достаточно. Также рассматривается partitioning внутри БД (например, партиционировать таблицу `ledger` по дате, если записей очень много).
- **Кэширование и CDN:** уменьшение нагрузки достигается и немасштабированием серверов, а снижением потребности в них. К примеру, внедрение CDN и кэширование на клиентской стороне для статики, эффективное использование Redis-кэша для повторяющихся запросов. Чем меньше запросов реально доходит до БД, тем больше масштабируемость.
- **Очереди и асинхронность:** как отмечалось, использование очередей позволяет сглаживать пики нагрузки. Если вдруг массово поступило 1000 заказов за секунду, они спокойно выстроятся в очередь и будут обработаны постепенно, не упав от одновременного запуска 1000 потоков. В это же время фронт API быстро отвечает пользователям о принятии заказа, не дожидаясь исполнения.
- **Monitoring и auto-scale:** важной частью масштабируемости является мониторинг производительности. Настраивается сбор метрик (CPU, время отклика, длина очередей, число активных заказов), на основе которых можно автоматически добавлять ресурсы. Например, если длина очереди заказов > X, автоматически поднять еще воркер; если QPS API достиг Y, добавить еще одну реплику веб-сервиса.
- **High Availability:** отказоустойчивость тоже аспект масштабируемости. Критичные компоненты (БД, Redis) нужно запускать в отказоустойчивой конфигурации: репликация, sentinel для Redis или кластер, резервные копии и пр. Микросервисы перезапускаются при сбое автоматически (например, через orkestrator). Это минимизирует простой системы под нагрузкой.
- **Тестирование на нагрузку:** перед ростом до целевых показателей проводятся нагрузочные тесты и профилирование, чтобы выявить узкие места. Затем адресно масштабируются или оптимизируются именно они.

Заложенные меры гарантируют, что система сможет плавно расти от небольшого старта до высокой нагрузки, оставаясь производительной. Фактически, аналогичные SMM-панели уже сейчас

обрабатывают тысячи заказов одновременно благодаря распределенной архитектуре [13], и Codex следует этим лучшим практикам.

7. Безопасность платформы

Безопасность – критичный аспект, особенно учитывая финансовые операции и открытый API. В архитектуру заложены следующие механизмы защиты:

- **Ограничение по IP:** для чувствительных частей системы вводятся ограничения доступа по IP-адресам. Например, админ-панель может быть доступна только с внутренних IP компании/VPN. Для API-ключей реселлеров реализуется опциональная привязка ключа к одному или нескольким IP (или подсети) – таким образом, даже если ключ утечет, запросы с неопознанного адреса будут отклонены. Поля `allowed_ips` в таблице `api_keys` хранят эти ограничения.
- **Rate Limiting (ограничение частоты запросов):** на уровне API-gateway или каждого сервиса действует ограничение на количество запросов в единицу времени от одного клиента (IP или API-ключа). Это предотвращает как случайные перегрузки, так и злонамеренные попытки DDoS или перебора ключей. Например, можно разрешить не более 5 запросов в секунду на чтение статуса заказа и не более 1 запроса в секунду на создание заказа с одного API-ключа (конкретные значения настраиваются). Реализация может быть с помощью in-memory counters или Redis (для распределенного окружения) – при каждом запросе инкремент, проверка и отклонение с кодом 429 Too Many Requests при превышении лимита [2]. Также можно вводить глобальные лимиты (не более N одновременных заказов в обработке у одного пользователя, чтобы предотвратить спам-создание).
- **Защита API-ключей:** ключи – как упомянуто – хранятся в безопасном виде (хеш или шифровка). При отображении в интерфейсе показываются только один раз при создании [5]. Пользователь может в любой момент сгенерировать новый ключ, а старый отозвать (ротация ключей). На стороне сервера при поступлении запроса с API-ключом всегда используется **HTTPS** (обязательно, HTTP не допускается, чтобы ключ не передавался в открытом виде). Можно предусмотреть длину ключа не меньше 32 символов с достаточной энтропией, чтобы перебор был нерентабелен. Дополнительно, **не** передавать ключи через URL (GET-параметры), только через заголовки или POST-поля, чтобы не логировать их случайно. В логах сервера ключи маскируются.
- **Аутентификация и авторизация:** все критичные действия требуют валидного токена/ключа. Реализована централизованная проверка JWT на каждое обращение (middleware) и проверка ролей. Таким образом, даже если кто-то получит токен обычного пользователя, он не сможет вызвать методы админов. Принцип наименьших привилегий: по умолчанию у токена пользователя только доступ к его данным, а для изменения чужих или системных настроек нужны права администратора.
- **Шифрование данных:** вся передача данных идет по HTTPS (TLS). Для хранения, PostgreSQL может шифровать данные на уровне диска (TDE) или как минимум настроены регулярные бэкапы с шифрованием. Особо чувствительные данные (например, приватные ключи провайдеров, API-ключи платежных систем) хранятся в зашифрованном виде или в защищенном хранилище (vault) с ограниченным доступом.
- **Защита от SQL-инъекций и XSS:** используется параметризированные запросы или ORM, чтобы исключить SQL-инъекции. Весь ввод, поступающий через API, валидируется (проверяются форматы, длины, недопустимые символы). Вывод данных, попадающий в UI,

экранируется чтобы предотвратить XSS. Эти задачи обычно решаются на уровне фреймворка, но архитектурно заложены требования к безопасному кодированию.

- **Audit Logging:** система ведет журнал важных действий: входы пользователей (кто, откуда, когда), действия администраторов (изменил баланс пользователя X, отменил заказ Y). Эти логи хранятся и доступны для проверки, что помогает расследовать подозрительную активность.
- **Защита от бот-атак:** публичные формы (регистрация, вход) могут быть защищены капчей, чтобы избежать массового создания фейковых аккаунтов или перебора паролей. При многократных неудачных попытках входа аккаунт временно блокируется (Brute-force protection).
- **Изоляция окружений:** отдельные среды для разработки/тестирования и боевой, чтобы тестовые ключи не пересекались с реальными. Также, можно настроить *rate limit* и для провайдеров – чтобы в случае, если у нас баг, мы не ушли в бесконечный цикл запросов к API поставщика и не получили бан (например, не делать более 1 запроса статуса в N секунд на одного провайдера, чтобы соответствовать их ограничениям).
- **Обновления и патчи:** поддерживается актуальность зависимостей, своевременно применяются обновления безопасности ОС, базы данных и прочего. Это организационная мера, но важная для архитектуры безопасности.

Сочетание этих мер обеспечивает многоуровневую защиту: от сети (HTTPS, IP whitelist) до приложений (ввод данных, права) и данных (шифрование, хранение ключей). Такие подходы соответствуют отраслевым практикам, в том числе требованиям стандартов для финансовых систем [15].

8. API для реселлеров

Для партнеров-реселлеров, которые хотят использовать платформу Codex как поставщика услуг, предоставляется специальный программный интерфейс – **API для реселлеров**. Он позволяет автоматизировать размещение заказов и интегрировать наш сервис в их собственные панели. Ниже описаны особенности этого API:

- **Аутентификация через API-ключ:** доступ осуществляется по REST API (HTTP POST-запросы) с использованием уникального ключа. Ключ передается либо в теле запроса, либо в заголовке (например, `Authorization: ApiKey XXXXX`). Каждый реселлерский ключ связан с определенным пользователем в нашей системе и имеет те же ограничения баланса. Запросы без действительного ключа или с недопустимого IP отклоняются. Управление ключами (генерация/отзыв) – через личный кабинет, как описано ранее. **Структура ключа:** длинная строка (например, 32 символа [0-9A-Za-z]) без предсказуемых шаблонов. Ключ может содержать префикс (для удобства идентификации, напр. `COD-xyz...`) и суффикс-контрольную сумму для валидации формата, но это вторично [16]. Главное – непредсказуемость и уникальность.
- **Методы API:** предоставляются основные методы, аналогичные пользовательскому функционалу, но адаптированные для автоматического использования. В частности:
- **Получение списка услуг:** реселлер может запросить полный перечень доступных услуг с параметрами (ID, название, описание, цены и лимиты). Это нужно, чтобы они синхронизировали свой каталог с нашим. Метод: `GET /api/v1/services` (или POST с action=services) с API-ключом, возвращает JSON массив услуг. *(Пример: запрос с параметром* `action=services` *и своим ключом вернет список услуг)* [17].

- **Создание заказа:** позволяет реселлеру разместить заказ от имени своего клиента. Требует указать идентификатор услуги (соответствует списку услуг), количество, и целевой объект (ссылку/ID). Метод: `POST /api/v1/order` или аналогично, передавая параметры заказа. В ответ возвращается внутренний ID заказа в нашей системе (для дальнейшего отслеживания) и статус (обычно "processing" если успешно принят). Если баланса недостаточно или параметры неверны – возвращается ошибка с пояснением. *(Пример: параметры запроса включают* `key=<API_KEY>`, `action=add`, `service=<ID услуги>`, `link=<URL или ID целевого аккаунта>`, `quantity=<кол-во>` *и пр.; успешный ответ содержит новый* `order` *ID)* [18]. Реселлер затем может сохранять этот ID у себя для проверки статуса.
- **Проверка статуса заказа:** реселлер может опрашивать статус ранее созданного заказа. Метод: `GET /api/v1/status` с параметрами ключа и ID заказа, возвращает текущий статус ("In progress", "Completed", "Partial (500/1000)" и т.п.) и дополнительные данные (количество выполнено, остаток, может быть ссылка на отчет). *(Пример: запрос* `action=status&order=<order_id>` *вернет структуру с полями статуса)* [11]. Также возможен метод массового запроса статусов нескольких заказов сразу (batch).
- **Проверка баланса:** чтобы реселлер мог узнавать, сколько средств у него осталось на счету (для пополнения вовремя), предоставляется метод `GET /api/v1/balance`. Он возвращает текущий баланс и валюту. *(Пример:* `action=balance` *с API-ключом – вернет сумму на балансе пользователя)* [19]. Пополнение баланса происходит вне API (через кабинет или по договоренности вручную), API только читает баланс.
- **Отмена заказа:** опционально, если поддерживается, метод для попытки отменить ранее сделанный заказ (в случае, если он еще не начал выполняться провайдером). Реализуется как `POST /api/v1/cancel` с указанием order_id. Возвращает успешность отмены (например, "Canceled" или ошибку "Already processing").

- **Дополнительные методы:** в перспективе можно поддержать метод **refill** (дозаказ, если списались подписчики, для услуг с гарантией) – некоторые панели поддерживают `action=refill` с указанием order_id, тогда провайдер снова добавляет немного услуг если упали. Также метод **get order** (получить подробности одного заказа) – хотя статус это уже дает. Основной функционал покроет перечисленное выше.

- **Формат и протокол:** API работает по HTTPS, формат данных JSON. Для совместимости с распространенными SMM-панелями можно реализовать поддержку и формата запросов как у них (многие панели ожидают ключ и action в POST-параметрах). В нашем случае, гибкость: можно принимать и URL-энкодед POST, и JSON body. Документация будет предоставлена, чтобы реселлеры быстро подключились.

- **Ограничения и квоты:** помимо глобальных rate limit, для реселлерского API могут быть установлены квоты: например, не более X заказов в сутки с одного ключа (или суммарно у реселлера тариф с лимитом). Также мониторится аномальная активность – если ключ генерирует подозрительный трафик (возможна компрометация), он может быть автоматически приостановлен.
- **Отчетность для реселлера:** хотя не метод API, но через кабинет или по запросу можно выгружать статистику использования API: сколько заказов создано, на какую сумму, оставшийся баланс, чтобы реселлеры могли сверять с своими системами.
- **Безопасность:** все перечисленные в разделе 7 меры относятся и к API для партнеров. Особенно важно, что ключи – это единственный фактор auth для API, поэтому их хранение и

передача должны быть максимально защищены. Реселлерам рекомендуется хранить ключи в секрете, менять их периодически. Мы же со своей стороны обеспечиваем защиту на уровне платформы (хранение хеша, ограничения по IP, отслеживание аномалий).

Данный API делает из платформы Codex полноценный **SMM-сервер** для других панелей: реселлеры смогут автоматически перепродавать наши услуги, а мы – обрабатывать эти заказы так же, как и от прямых пользователей. Такой двухуровневый подход (поставщик–реселлер) широко используется в индустрии SMM [20] [21], что позволит масштабировать бизнес не только технически, но и по партнерской сети.

Заключение: Предложенная архитектура охватывает все ключевые компоненты SMM-маркетплейса – от обработки заказов и платежей до интеграции с внешними провайдерами и предоставления API для партнеров. Она построена с учетом будущего роста и высокой нагрузки: микросервисный подход, очереди для асинхронности и масштабирование по горизонтали обеспечат обработку тысяч заказов в сутки без деградации производительности [22]. Особое внимание уделено безопасности (защита финансовых данных, API-ключей, ограничение доступа) и гибкости (легкость добавления новых услуг, провайдеров, модулей). Эта архитектура послужит надежным фундаментом для развития платформы Codex в успешный и устойчивый SMM-маркетплейс.

---

[1] [4] [7] [10] Как работают SMM-панели? Разбор изнутри : r/SocialMediaMarketing
https://www.reddit.com/r/SocialMediaMarketing/comments/1jvztmk/how_do_smm_panels_work_a_behindthescenes_breakdown/?tl=ru

[2] [8] [14] [15] Building a Microservice-Based Payment Wallet Architecture
https://www.linkedin.com/pulse/building-microservice-based-payment-wallet-meenakshi-kalia-mgmyc

[3] [6] [9] [13] [20] [21] [22] Как работают SMM-панели? | Ecommerce Fastlane
https://ecommercefastlane.com/ru/%D0%9A%D0%B0%D0%BA-%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%B0%D1%8E%D1%82-SMM-%D0%BF%D0%B0%D0%BD%D0%B5%D0%BB%D0%B8/

[5] [12] [16] API Key Authentication Best Practices | Zuplo Blog
https://zuplo.com/blog/api-key-authentication

[11] [17] [18] [19] SMM Panel - API Documentation
https://glorysmmpanel.com/api