

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
вычислительной техники

Кафедра
МОиПЭВМ

Направление подготовки 09.03.04 «Программная инженерия»

Направленность Программная инженерия

БАКАЛАВРСКАЯ РАБОТА

на тему

**Разработка бэкенд-компонента информационной
системы компьютерной фирмы**

Студент _____ Давыдов Константин Алексеевич
(подпись, дата) (ФИО полностью)

Руководитель _____ Самуйлов С. В.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр _____ Такташкин Д.В.
(подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от _____ № _____)

Заведующий кафедрой _____ Козлов А.Ю.
(подпись) (фамилия, инициалы)

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____ № _____)

Секретарь ГЭК _____ Попова Н.А.
(подпись) (фамилия, инициалы)

Пенза, 2026

Федеральное государственное бюджетное образовательное учреждение
высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет

Вычислительной техники

Кафедра

МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой
А.Ю. Козлов
«26» декабря 2025 г

**ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА**

1. Студент Давыдов Константин Алексеевич гр. 22ВП2 факультета ВТ
<Фамилия, имя, отчество полностью>

2. Тема работы Разработка бэкенд-компонента информационной системы
компьютерной фирмы

Тема утверждена приказом ПГУ № _____ от " _____ " _____ 20 _____

3. Руководитель работы _____ к.т.н., доцент, доцент кафедры «МОиПЭВМ»
Самуйлов С. В.

4. Задание на работу (назначение разработки, исходные данные и т.п.)

1) Назначение: серверная часть интернет-магазина электроники и услуг
(e-commerce + услуги ремонта/сборки ПК).

2) Среда выполнения: Linux Ubuntu 22.04, Windows 10/11, macOS.

3) Языковые средства разработки: Python 3.13, SQL, UML, YAML.

4) Инструментальные средства: FastAPI 0.136, SQLAlchemy 2.0, Alembic,
Pydantic 2, aiokafka, redis-py, httpx, structlog, uv 0.10, PostgreSQL 18, Redpanda 25
(Kafka API), MinIO, Docker, Visual Paradigm 17.

5) Технологии разработки: микросервисная архитектура, event-driven
choreography saga, outbox pattern, CQRS, CloudEvents 1.0, асинхронный ввод-вывод
(asyncio).

6) Основные функции приложения:

а) регистрация и авторизация с подтверждением по e-mail;

б) управление каталогом товаров и категорий через CMS;

в) корзина в Redis;

г) оформление заказа с гарантированной идемпотентностью;

д) оплата через хореографическую сагу;

е) приём заявок на услуги;

ж) интеграция с системой уведомлений;

з) интеграция со встроенным поиском.

5. Перечень подлежащих разработке вопросов:

1) Анализ предметной области и постановка задачи;

2) Проектирование;

3) Реализация;

4) Контроль качества программного обеспечения;

5) Оформление пояснительной записки.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области и требований к системе	10%	07.04.2026 – 13.04.2026	
2	Проектирование программных средств	20%	14.04.2026 – 25.04.2026	
3	Реализация программных средств	30%	27.04.2026 – 11.05.2026	
4	Тестирование программных средств	20%	12.05.2026 – 21.05.2026	
5	Оформление пояснительной записки	20%	22.05.2026 – 02.06.2026	

Дата выдачи задания «26» декабря 2025г.

Руководитель бакалаврской работы _____
(подпись, дата)

Задание к исполнению принял студент _____
(подпись, дата)

Работу к защите допустить
Декан ФВТ _____ Л.Р. Фионова
(подпись, дата)

Реферат

Пояснительная записка содержит 81 страницу, 22 рисунка, 13 таблиц, 20 использованных источников и 9 приложений.

ИНТЕРНЕТ-МАГАЗИН, МИКРОСЕРВИСНАЯ АРХИТЕКТУРА, FASTAPI, POSTGRESQL, KAFKA, REDIS, CLOUDEVENTS, ХОРЕОГРАФИЧЕСКАЯ САГА, OUTBOX PATTERN, ИДЕМПОТЕНТНОСТЬ, ВНУТРЕННИЙ КОШЕЛЁК, ВЕБ-ПРИЛОЖЕНИЕ.

Объект исследования – серверная часть интернет-магазина электроники, обслуживающего операции оформления заказов, проведения платежей через внутренний кошелёк пользователя и рассылки транзакционных уведомлений.

Цель выпускной квалификационной работы – спроектировать и разработать серверное программное обеспечение, обеспечивающее полный жизненный цикл электронной коммерции: от просмотра каталога до оплаты заказа с внутреннего кошелька пользователя.

Технологии разработки – язык Python 3.13 в среде Visual Studio Code, FastAPI 0.136 поверх Uvicorn. Единая СУБД PostgreSQL 18. Кеш, корзина и идемпотентность – Redis 7. Брокер Redpanda 25, совместимый с Kafka API, по контрактам CloudEvents 1.0 с типизацией через Pydantic. Развёртывание автоматизировано Docker Compose.

Среда исполнения – Linux-сервер с установленным Docker.

Результаты работы – в ходе выполнения выпускной квалификационной работы была спроектирована и реализована серверная часть интернет-магазина из шести микросервисов, выполняющая сквозную хореографическую сагу оформления и оплаты заказа с внутреннего кошелька.

					ПГУ 09.03.04 – 08BP222.07 ПЗ						
Изм.	Лист	№ докум.	Подпись	Дата							
Разраб.		Давыдов К. А.			Разработка бэкенд-компонента информационной системы компьютерной фирмы	Лит.		Лист	Листов		
Провер.								4	81		
Реценз.						ГРУППА 22BP2					
Н. Контр.		Такташкин Д.В.									
Утверд.		Козлов А.Ю									

Содержание

Введение.....	7
1 Анализ предметной области и постановка задачи на разработку серверной части	8
1.1 Анализ предметной области приложения	8
1.2 Анализ архитектурных паттернов распределённой обработки заказов	10
1.3 Анализ аналогичных программных средств	11
1.4 Постановка задачи на разработку серверной части.....	14
1.4.1 Цели и задачи приложения.....	14
1.4.2 Классы пользователей интернет-магазина	15
1.5 Анализ требований.....	16
1.5.1 Анализ функциональных требований	16
1.5.2 Анализ нефункциональных требований	24
1.6 Планирование разработки и оценка бюджета	25
2 Проектирование серверной части интернет-магазина	28
2.1 Архитектура приложения.....	28
2.2 Проектирование структур данных	30
2.2.1 Концептуальная модель баз данных	30
2.2.2 Логическая модель баз данных.....	33
2.3 Обобщённая структура разрабатываемого сервиса.....	36
2.4 Требования к аппаратному обеспечению серверной части	37
3 Разработка программного обеспечения серверной части	39
3.1 Разработка архитектуры серверной части	39

	6
3.2 Разработка физической модели базы данных	44
3.3 Разработка архитектурных слоёв внутри микросервисов	47
3.3.1 Слой представления данных	48
3.3.2 Слой бизнес-логики	49
3.3.3 Слой обращения к базе данных	50
4 Контроль качества программного обеспечения.....	52
4.1 Обоснование выбора модели тестирования и метрик кода	52
4.2 Тестирование серверной части интернет-магазина.....	52
4.2.1 Функциональное тестирование саги оплаты заказа	52
4.2.2 Модульное тестирование обработчика веб-хука	54
4.2.3 Метрики Холстеда метода создания заказа.....	55
Заключение	57
Список использованных источников	58
Приложение А (справочное) Глоссарий	60
Приложение Б (обязательное) Объекты обращения к базам данных	63
Приложение В (обязательное) SQL-скрипты создания таблиц.....	65
Приложение Г (обязательное) Структуры слоя представления данных	68
Приложение Д (обязательное) Структуры слоя бизнес-логики.....	70
Приложение Е (обязательное) Структуры слоя обращения к базе данных	73
Приложение Ж (обязательное) Листинг модульных тестов.....	75
Приложение И (рекомендуемое) Спецификация атрибутов таблиц	77
Приложение К (рекомендуемое) Иллюстрационные материалы.....	79

Введение

Российский рынок электронной коммерции стремительно растет, достигнув в 2024 году объема свыше 9 трлн рублей [1]. Это требует отказа от монолитных решений в пользу гибких систем, способных выдерживать пиковые нагрузки и обеспечивать быструю разработку. Важным элементом современных площадок становятся внутренние кошельки для реализации программ лояльности, функционирование которых требует применения микросервисной архитектуры и асинхронной шины событий [2].

Основная техническая сложность заключается в синхронизации данных между независимыми сервисами при оплате заказа. Для предотвращения ошибок, таких как двойные списания или потеря данных, применяется паттерн «хореографическая сага», позволяющий микросервисам автономно координировать действия через брокер сообщений [3]. Целью данной работы является проектирование и разработка отказоустойчивой серверной части магазина электроники с внутренним кошельком, гарантирующей идемпотентность и целостность транзакций.

В процессе реализации были проанализированы архитектурные паттерны, спроектированы изолированные базы данных и контракты событийного обмена. Итогом стала программная реализация шести микросервисов на FastAPI с использованием механизмов Transactional Outbox и саги оплаты. Проведенное тестирование и расчет метрик Холстеда подтвердили высокое качество кода и стабильность системы. Результат работы представляет собой готовую модель серверной инфраструктуры, имеющую практическую ценность для бизнеса и образовательного процесса.

1 Анализ предметной области и постановка задачи на разработку серверной части

1.1 Анализ предметной области приложения

Работа посвящена вопросам построения серверных приложений для e-commerce, которые сопровождают процесс покупки на всех его стадиях. Для глубокого анализа проблем разработки программных продуктов необходимо прежде всего систематизировать ключевые термины, характерные для этой сферы деятельности.

Электронная коммерция (e-commerce) – отрасль экономики, связанная с продажей товаров и услуг через интернет, включающая операции оформления заказа, оплаты, доставки и послепродажного обслуживания [1].

Интернет-магазин – программное обеспечение, реализующее процессы электронной коммерции с точки зрения конкретного продавца: каталог товаров, корзина покупателя, оформление заказа, приём платежей, личный кабинет и обработка возвратов.

Микросервисная архитектура – стиль построения распределённых приложений, при котором система собирается из небольших независимо развёртываемых сервисов, каждый из которых отвечает за свою предметную область и общается с остальными через сеть по стандартизированным контрактам [4].

Сага (saga) – последовательность локальных транзакций в нескольких сервисах, связанных событиями: успешное завершение каждой транзакции порождает событие, запускающее следующую; при ошибке выполняются компенсирующие действия, возвращающие систему в согласованное состояние [3].

Шаблон transactional outbox (исходящих сообщений) – приём, гарантирующий, что сохранение бизнес-данных в локальной БД и публикация

события о них в брокере сообщений происходит атомарно: событие сначала записывается в специальную таблицу `outbox` в той же транзакции, а отдельный процесс-релей публикует его в брокер [3].

`CloudEvents 1.0` – открытая спецификация CNCF, описывающая стандартный конверт события с обязательными атрибутами ``specversion``, ``id``, ``source``, ``type``, ``time`` и опциональным ``datacontenttype``. Позволяет передавать события между разными технологиями и языками без преобразования формата [5].

Идемпотентность – свойство операции, при котором её повторное выполнение с теми же параметрами приводит к тому же результату, что и однократное. Является обязательным требованием для всех точек входа в распределённую систему, так как сетевые сбои и повторные попытки клиентов делают невозможным гарантию ровно-однократной доставки.

Внутренний кошелёк пользователя (`wallet`) – учётная запись внутреннего баланса клиента в денежном выражении, не привязанная к внешним платёжным системам. Используется как один из методов оплаты заказа наряду с банковской картой и СБП. На кошелёк начисляются: приветственный бонус при регистрации, возвраты по отменённым заказам, маркетинговые подарочные суммы. Программа лояльности на основе внутреннего кошелька является типовой механикой удержания клиентов в современной розничной электронной коммерции.

Таким образом, анализ предметной области показывает, что разрабатываемое серверное приложение должно выступить связующим звеном между двумя группами пользователей – покупателями и администраторами – обслуживая операции электронной коммерции с использованием внутреннего кошелька в качестве одного из методов оплаты, при этом обеспечивая одновременно высокую скорость отклика, согласованность данных и расширяемость под рост ассортимента и нагрузки.

1.2 Анализ архитектурных паттернов распределённой обработки заказов

Ключевая операция любого интернет-магазина – оформление и оплата заказа – в микросервисной архитектуре пересекает границы как минимум трёх сервисов: «заказа», «платежа» и «кошелька» (для списания средств с внутреннего баланса). Прямая распределённая транзакция через двухфазный коммит (2PC) считается антипаттерном в современных облачных архитектурах из-за блокировок, низкой производительности и плохой переносимости между СУБД [3]. Вместо этого предметная область требует анализа альтернативных паттернов: orchestration-саги, choreography-саги и эпизодической согласованности через event sourcing.

Для сравнения паттернов введены следующие критерии: связанность сервисов (должен ли один сервис знать о существовании других); наличие единой точки отказа (orchestrator); простота добавления нового шага; сложность отладки и трассировки; возможность переиспользования событий. В таблице 1 приведено сравнение архитектурных паттернов распределённой обработки.

Таблица 1 – Сравнение архитектурных паттернов распределённой обработки

Критерий	Orchestration-сага	Choreography-сага	Event sourcing
Связанность сервисов	Высокая (через orchestrator)	Низкая (через события)	Низкая
Единая точка отказа	Да – orchestrator	Нет	Нет (требуется event store)
Добавление нового шага	Изменить orchestrator	Подписать новый consumer	Подписать новый projector
Сложность отладки	Средняя	Высокая (нужны корреляция и трассировка)	Высокая
Стоимость	Средняя	Низкая	Высокая

Продолжение таблицы 1

Критерий	Orchestration-сaga	Choreography-сaga	Event sourcing
инфраструктуры			
Подходит для проекта	Частично	Полностью	Избыточно

По итогам сравнения в качестве основного паттерна для разрабатываемого серверного приложения выбрана choreography-сaga. Её ключевые преимущества – отсутствие единой точки отказа и возможность независимого развёртывания новых consumer'ов – соответствуют принципам микросервисной архитектуры. Сложность отладки компенсируется применением сквозной трассировки через атрибут `ce\_id` конверта CloudEvents и единой системой структурного логирования через structlog.

Для обеспечения атомарности «записать данные → опубликовать событие» применяется transactional outbox. Альтернатива – двухфазная публикация через распределённую транзакцию между БД и брокером – была отклонена из-за низкой производительности и ограниченной поддержки в выбранных технологиях.

Для гарантирования идемпотентности входных точек (создание заказа, обработка веб-хука платёжной системы) использованы две дополняющие техники: семантический ключ идемпотентности на уровне HTTP-заголовка Idempotency-Key с проверкой через SETNX в Redis; ограничение UNIQUE в Postgres как защита от гонок при отказе Redis. Эта пара обеспечивает корректное поведение и при повторных запросах клиента, и при повторной доставке события из Kafka.

1.3 Анализ аналогичных программных средств

Для оценки текущего состояния отрасли и поиска лучших практик был проведён анализ публично доступной информации по серверной архитектуре российских интернет-магазинов электроники и универсальных маркетплейсов:

Ozon, DNS-shop, Wildberries, М.Видео-Эльдорадо. Анализ опирался на инженерные блоги, доклады с конференций Highload++/Saint Highload, GitHub-репозитории open-source-компонентов и публичные технические радары. В таблице 2 приведены общие сведения об аналогах.

Таблица 2 – Общие сведения об аналогах

Платформа	Стек серверной части	Описание
Ozon	Go + PostgreSQL + Kafka + Vitess	Универсальный маркетплейс. Серверное ядро написано на Go, активно используется Kafka для event-driven-интеграций, шардирование MySQL/Vitess для хранения заказов.
DNS-shop	C# .NET + MS SQL + RabbitMQ	Сеть магазинов электроники. Сильная интеграция с офлайн-точками и собственной логистикой; реализована классическая трёхзвенная архитектура с шиной RabbitMQ.
Wildberries	Go + PostgreSQL + Kafka + ClickHouse	Крупнейший маркетплейс одежды и электроники. Активно использует Kafka для обновления каталога и ClickHouse для аналитики. Известны публичные доклады об outbox и собственной системе доставки сообщений.
М.Видео-Эльдорадо	Java/Kotlin + Oracle + Kafka	Объединённая розничная сеть. Серверная часть унаследована от Oracle Commerce; ведётся миграция на микросервисы на Spring Boot и Kafka.

Каждое из перечисленных решений обладает собственными сильными и слабыми сторонами. К общим сильным сторонам относятся: высокая горизонтальная масштабируемость серверной части, обширная инструментальная поддержка процессов CI/CD, развитые системы мониторинга

и логирования. К общим недостаткам – закрытость стека (нет возможности изучить внутреннюю архитектуру), тяжеловесность для небольших розничных проектов, привязка к коммерческим СУБД.

Для проведения количественного сравнения выделены критерии: открытость стека; поддержка хореографических sag; применение outbox-pattern; типизация событий (схема-реестр); интеграция с поисковым движком; поддержка приёма заявок на услуги, не только товаров; публичная документация по архитектуре.

В таблице 3 представлен сравнительный анализ аналогов. Шкала оценивания: 0 баллов – функционал отсутствует или не подтверждён; 1 балл – реализован частично; 2 балла – полностью реализован и публично описан.

Таблица 3 – Сравнительный анализ аналогов

Критерий	Ozon	DNS-shop	Wildberries	М.Видео-Эльдорадо
Открытость стека	1	0	1	0
Поддержка choreography-sag	2	1	2	1
Применение outbox-pattern	2	1	2	1
Типизация событий (schema registry)	1	1	2	1
Интеграция с поисковым движком	2	1	2	1
Внутренний кошелёк / программа лояльности	2	1	2	2
Публичная архитектурная документация	2	1	2	1
Итого	12	6	12	8

Из анализа следует, что лидирующие решения (Ozon, Wildberries) полностью реализуют choreography-saги, outbox-pattern и собственный механизм внутреннего баланса с кешбэк-программой, что подтверждает обоснованность выбора этих паттернов для разрабатываемого приложения.

Одновременно ни одно из решений не предоставляет публично документированную микросервисную архитектуру с типизацией CloudEvents и открытой реализацией кошелька, что обосновывает необходимость собственной разработки.

1.4 Постановка задачи на разработку серверной части

1.4.1 Цели и задачи приложения

Основной целью выпускной квалификационной работы является разработка серверной части программного обеспечения интернет-магазина электроники. Главной функцией приложения должно стать обслуживание полного жизненного цикла заказа: от просмотра каталога и формирования корзины до оплаты заказа с внутреннего кошелька пользователя и доставки уведомления. Помимо стандартной товарной части, серверное приложение должно реализовывать программу лояльности через внутренний кошелек: начисление приветственного бонуса при регистрации, возврат средств на баланс при отмене заказа, идиempotentное списание баланса при оплате.

К основным задачам, решаемым серверной частью, относятся:

- регистрация и авторизация пользователей с подтверждением e-mail и ролевой моделью доступа;
- ведение каталога товаров и категорий через систему управления контентом (Payload CMS);
- обслуживание персональной корзины покупателя в Redis с TTL для устаревших корзин;
- оформление заказа с гарантией идиempotentности на уровне HTTP-запроса (Idempotency-Key);
- проведение оплаты заказа через хореографическую сагу с переходом в состояние «failed» при отказе;

- обслуживание внутреннего кошелька пользователя: учёт баланса, идемпотентное списание (debit), возврат (refund), приветственный бонус (signup_bonus);
- service-to-service-вызов кошелька из payment-service с защитой через X-Internal-Token;
- формирование и отправка транзакционных уведомлений (e-mail) через консьюмер Kafka-топиков;
- формирование CQRS read-projection саги в таблице order_events для отображения временной линии заказа в личном кабинете;
- интеграция с публичной API-точкой доступа (Storefront BFF) и административным интерфейсом (CMS).

Результатом выполнения этих задач должна стать распределённая система из шести микросервисов (api-gateway, auth-service, order-service, payment-service, wallet-service, notification-worker), обменивающихся событиями через брокер сообщений по контрактам CloudEvents 1.0, сохраняющая бизнес-данные в собственных базах PostgreSQL и опирающаяся на единое объектное хранилище MinIO для медиа.

1.4.2 Классы пользователей интернет-магазина

Для разрабатываемого интернет-магазина выделены три класса пользователей: покупатель, контент-менеджер и администратор. В таблице 4 подробно описаны их бизнес-процессы и затрагиваемые сущности данных.

Таблица 4 – Классы пользователей и их бизнес-процессы

№	Класс	Часть сервиса	Бизнес-процессы	Используемые объекты
1	Покупатель	Storefront (Next.js + BFF)	Регистрация и авторизация. Просмотр каталога. Управление корзиной.	User, Cart, Order, Payment, Wallet, WalletTx

Продолжение таблицы 4

№	Класс	Часть сервиса	Бизнес-процессы	Используемые объекты
			Оформление заказа. Оплата с внутреннего кошелька. Просмотр баланса и истории операций. Отслеживание статуса заказа. Получение уведомлений	
2	Контент-менеджер	Admin (Payload CMS)	Создание и публикация товаров. Управление категориями. Загрузка медиа. Редактирование баннеров и информационных страниц	Product, Category, Media, Banner
3	Администратор	Admin (Payload CMS) + Storefront	Управление пользователями и ролями. Просмотр и обработка всех заказов. Корректировка балансов кошельков пользователей (refund). Контроль метрик системы	User, Order, Wallet, WalletTx

Разрабатываемое серверное приложение должно полностью покрывать перечисленные классы пользователей и все указанные бизнес-процессы.

1.5 Анализ требований

1.5.1 Анализ функциональных требований

Диаграмма вариантов использования (use case diagram) предоставляет верхнеуровневую карту функций, которые серверная часть должна поддерживать, и связывает их с классами пользователей. Согласно нотации UML 2.5, такая диаграмма не описывает внутреннее устройство системы, а отражает её внешнее поведение [6].

Исходя из определенных ранее ролей пользователей и требований проекта, была сформирована диаграмма прецедентов (Use Case). Она визуализирует такие функции, как регистрация и вход в систему, операции с товарами и корзиной, проведение платежей, мониторинг статуса доставки, а также инструменты администрирования (управление контентом, аккаунтами и процедурами возврата средств). Детальная проработка каждого кейса представлена в соответствующих спецификациях, где зафиксированы входные и выходные условия, а также алгоритмы выполнения (базовые и вариантные). Графическое исполнение диаграммы приведено на рисунке 1.

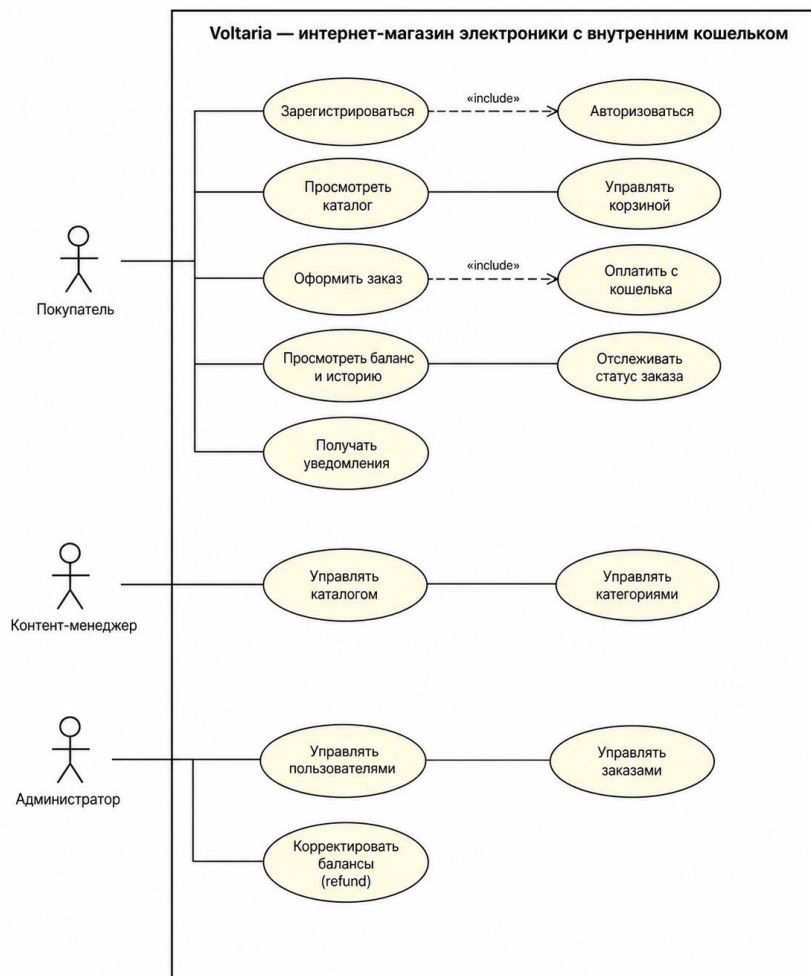


Рисунок 1 – Диаграмма вариантов использования

Для наиболее критичных сценариев – авторизации, оформления заказа и оплаты с внутреннего кошелька – разработаны диаграммы деятельности и

развёрнутые спецификации вариантов использования. Спецификация варианта использования «Авторизоваться» приведена в таблице 5.

Таблица 5 – Спецификация варианта использования «Авторизоваться»

Раздел	Содержание
Краткое описание	Пользователь предъявляет учётные данные и получает access-токен
Актёр	Покупатель (зарегистрированный)
Предусловия	Пользователь зарегистрирован, e-mail подтверждён, аккаунт не заблокирован
Основной поток	<ol style="list-style-type: none"> 1. Пользователь отправляет POST /api/v1/auth/login с e-mail и паролем. 2. auth-service применяет rate-limit (5 попыток в минуту). 3. Сервис вычисляет argon2id-хэш пароля с pepper и сравнивает с сохранённым. 4. При совпадении генерируется access-JWT и refresh-токен. 5. Refresh сохраняется в Redis под ключом user_id с TTL. 6. Клиент получает токены в httpOnly cookies
Альтернативные потоки	<ol style="list-style-type: none"> A1. E-mail не подтверждён → 403 с сообщением о необходимости подтверждения. A2. Неверный пароль → 401, увеличивается счётчик rate-limit. A3. Превышен rate-limit → 429
Постусловия	Установлены cookies, refresh зафиксирован в Redis

В рамках реализации механизма авторизации используется технология JWT-токенов, обеспечивающая возможность безопасной идентификации пользователя при последующих обращениях к системе. Хранение refresh-токенов в Redis с ограниченным временем жизни позволяет контролировать активные пользовательские сессии и обеспечивает возможность их принудительного завершения при необходимости.

Диаграмма деятельности для спецификации «авторизация» приведена на рисунке 2.

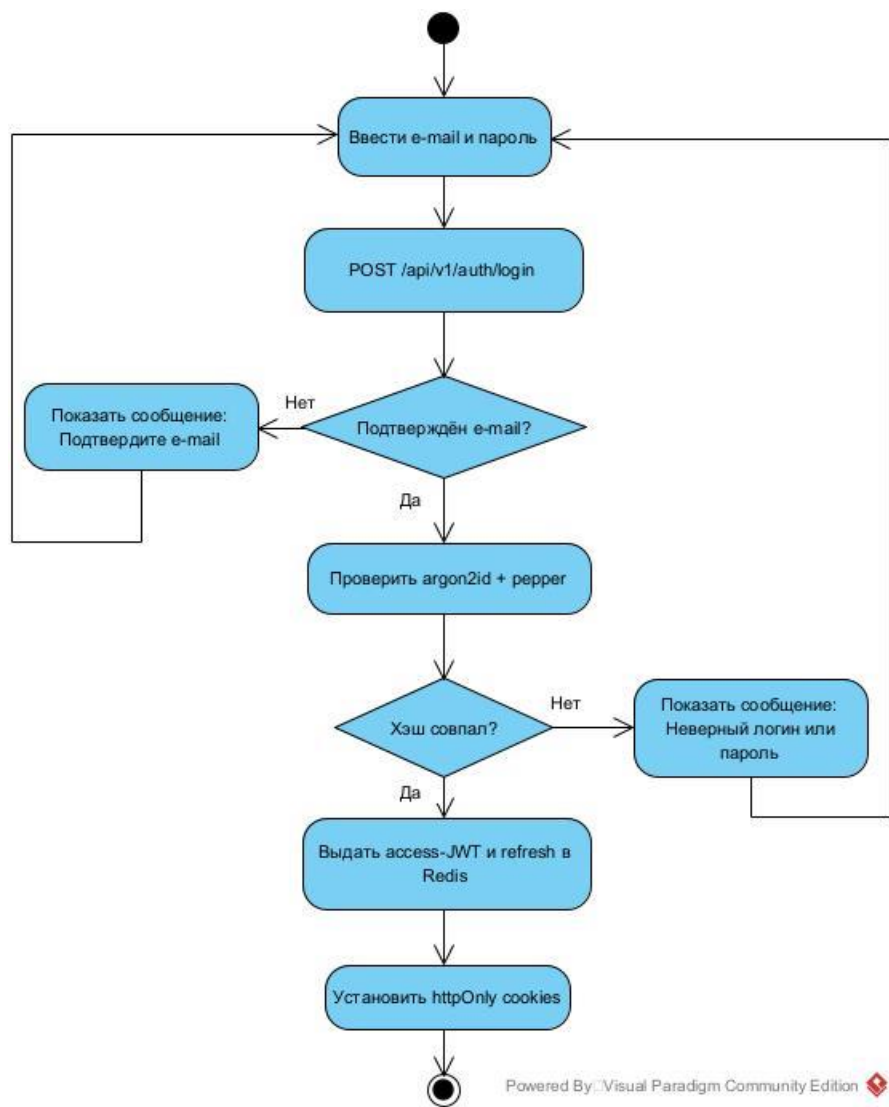


Рисунок 2 – Спецификация варианта использования «Авторизация»

В таблице 6 приведены спецификации варианта использования «Оформить заказ».

Таблица 6 – Спецификация варианта использования «Оформить заказ»

Раздел	Содержание
Краткое описание	Покупатель преобразует содержимое корзины в заказ в БД

Продолжение таблицы 6

Раздел	Содержание
Актёр	Покупатель
Предусловия	Пользователь авторизован. Корзина не пуста. Все товары имеют остатки
Основной поток	1. Клиент отправляет POST /api/v1/orders с заголовком Idempotency-Key и составом корзины. 2. order-service проверяет ключ в Redis через SETNX – занят/свободен. 3. Если ключ свободен – выполняется stock-проверка через Payload. 4. В одной транзакции создаётся запись orders, order_items и outbox-событие order.created. 5. Outbox-relay публикует событие в Kafka. 6. payment-service подписан на topic order.created и запускает сценарий оплаты
Альтернативные потоки	A1. Ключ уже занят → возвращается ранее созданный заказ (идемпотентность). A2. Недостаточно остатков → 409 Conflict, outbox не пишется. A3. Сбой Postgres → клиент получает 503, заказ не создан
Постусловия	Заказ создан в статусе pending, событие order.created опубликовано в Kafka

Для варианта использования «Оформить заказ» особое значение имеет обеспечение целостности данных и устойчивости системы к повторным запросам. Использование механизма идемпотентности на основе заголовка Idempotency-Key позволяет исключить создание дублирующихся заказов при повторной отправке запроса клиентом. Дополнительно выполняется проверка актуальности товарных остатков перед созданием заказа, что предотвращает возникновение конфликтных ситуаций при одновременном оформлении покупок несколькими пользователями.

На рисунке 3 отображена диаграмма деятельности варианта использования «Оформить заказ».

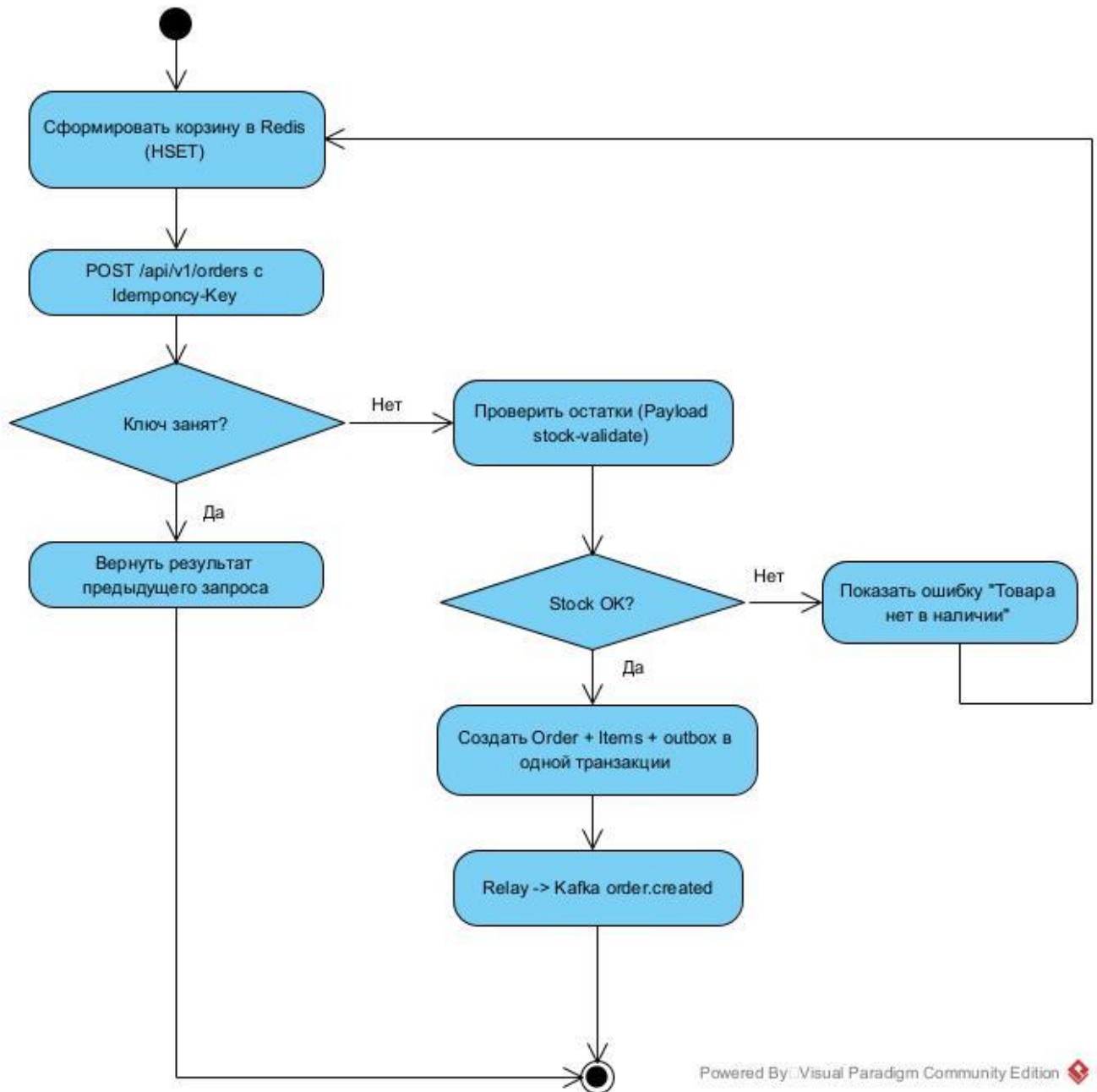


Рисунок 3 – Спецификация «Оформить заказ»

Представленная диаграмма деятельности отражает последовательность взаимодействия основных компонентов системы при создании заказа.

В таблице 7 приведены спецификации варианта использования «Оплатить с кошелька».

Таблица 7 – Спецификация варианта использования «Оплатить с кошелька»

Раздел	Содержание
Краткое описание	Хореографическая сага списания средств с внутреннего баланса для оплаты заказа
Актёр	Покупатель (инициатор), payment-service (исполнитель)
Предусловия	Заказ существует в статусе pending. Кошелёк пользователя инициализирован. Баланс $\geq$ суммы заказа
Основной поток	<ol style="list-style-type: none"> 1. payment-service получает событие order.created из Kafka. 2. Создаётся запись payments(status=pending, provider='wallet') в БД payment. 3. payment-service вызывает wallet-service POST /api/v1/wallet/debit с X-Internal-Token и Idempotency-Key=payment_id. 4. wallet-service блокирует строку wallets SELECT FOR UPDATE. 5. Создаётся wallet_transactions(kind='debit', amount, balance_after) и запись в outbox. 6. Транзакция фиксируется, возвращается 200 ОК с новым балансом. 7. payment-service переводит payment в status=succeeded и записывает payment.confirmed в outbox. 8. order-service подписан на payment.confirmed и переводит заказ в paid
Альтернативные потоки	<p>A1. Недостаточно средств → wallet возвращает 402, payment.status=failed, выпускается payment.failed, заказ переходит в cancelled.</p> <p>A2. Повторный вызов debit с тем же Idempotency-Key → wallet возвращает прежний результат.</p> <p>A3. Сетевой сбой wallet → payment-service ретраит с экспоненциальной задержкой</p>
Постусловия	Баланс кошелька уменьшен. Заказ в статусе paid. События

Продолжение таблицы 7

Раздел	Содержание
	wallet.debited и payment.confirmed опубликованы

На рисунке 4 отображена диаграмма деятельности варианта использования «Оплатить с кошелька».

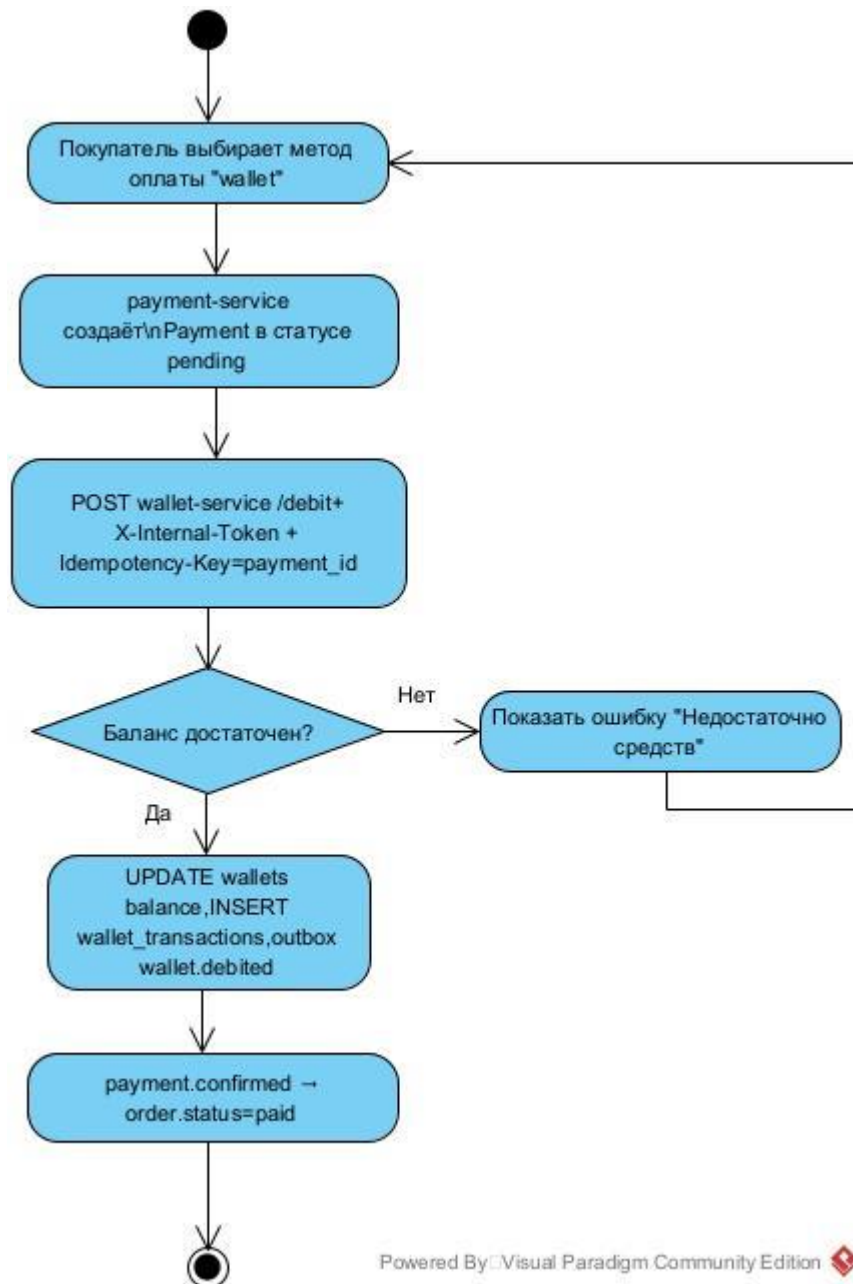


Рисунок 4 – Спецификация «Оплатить с кошелька»

Для обеспечения бесперебойного функционирования бизнес-логики интернет-магазина и удовлетворения потребностей всех категорий акторов был выделен ключевой набор сценариев. Их отбор производился путем тщательной систематизации требований, предъявляемых к функциональности системы.

1.5.2 Анализ нефункциональных требований

Помимо функциональных требований, для серверной части интернет-магазина сформулирован ряд нефункциональных требований, продиктованных спецификой e-commerce-нагрузки и особенностями выбранной архитектуры:

- гибкость – возможность расширять систему новыми сервисами без модификации существующих, обеспечивается контрактом CloudEvents и подпиской через consumer group;
- масштабируемость – независимое горизонтальное масштабирование загруженных компонентов достигается за счёт stateless-микросервисов и хранения состояния в общих БД;
- отказоустойчивость – частичный отказ одного микросервиса не должен парализовать систему, при этом обеспечивая ее изолированными базами данных и буферизацией событий в Kafka;
- производительность – медианное время отклика на запрос оформления заказа не должно превышать 250 мс; сквозная сага оплаты не более 1,5 с;
- идемпотентность – все API, изменяющие состояние, должны корректно обрабатывать повторные запросы клиентов и повторную доставку сообщений из Kafka;
- безопасность – пароли хранятся с использованием Argon2id и pepper; все коммуникации между публичной частью и серверной – поверх TLS; refresh-токены реализуют ротацию с детектированием попыток угона;

– наблюдаемость – единый формат структурированных логов в формате JSON, поддержка трассировки через атрибут `se\_id` события.

Перечисленные требования обоснованы спецификой e-commerce-приложения: обработка большого числа параллельных запросов, частые сбои сети при коммуникации с платёжным шлюзом и необходимость безостановочной выкатки нового функционала.

Опираясь на перечисленные требования, в качестве основного архитектурного стиля выбрана микросервисная архитектура: её модульность, независимость компонентов и возможность точечного масштабирования отдельных сервисов лучше всего подходят под характер нагрузки интернет-магазина.

1.6 Планирование разработки и оценка бюджета

Фундаментом для эффективного создания программных систем служит тщательное проектирование графиков работ и расчет финансовых затрат [7]. Процесс администрирования проекта осуществлялся в среде Project Libre, которая позволяет эффективно координировать задачи и распределять имеющиеся ресурсы. С целью визуализации временных рамок и последовательности действий была разработана диаграмма Ганта. Данный инструмент представляет собой графическую модель, в которой перечень проектных задач сопоставляется с временной шкалой [8]. Дополнительно следует отметить, что использование диаграммы Ганта позволяет более наглядно отслеживать текущее состояние выполнения проекта и своевременно выявлять возможные отклонения от запланированного графика. Это способствует повышению прозрачности процессов управления и облегчает взаимодействие между участниками команды. Кроме того, применение специализированных инструментов планирования, таких как Project Libre, обеспечивает более точное распределение ресурсов и снижает вероятность

возникновения организационных рисков. Итоговый график выполнения работ отображен на рисунке 5.

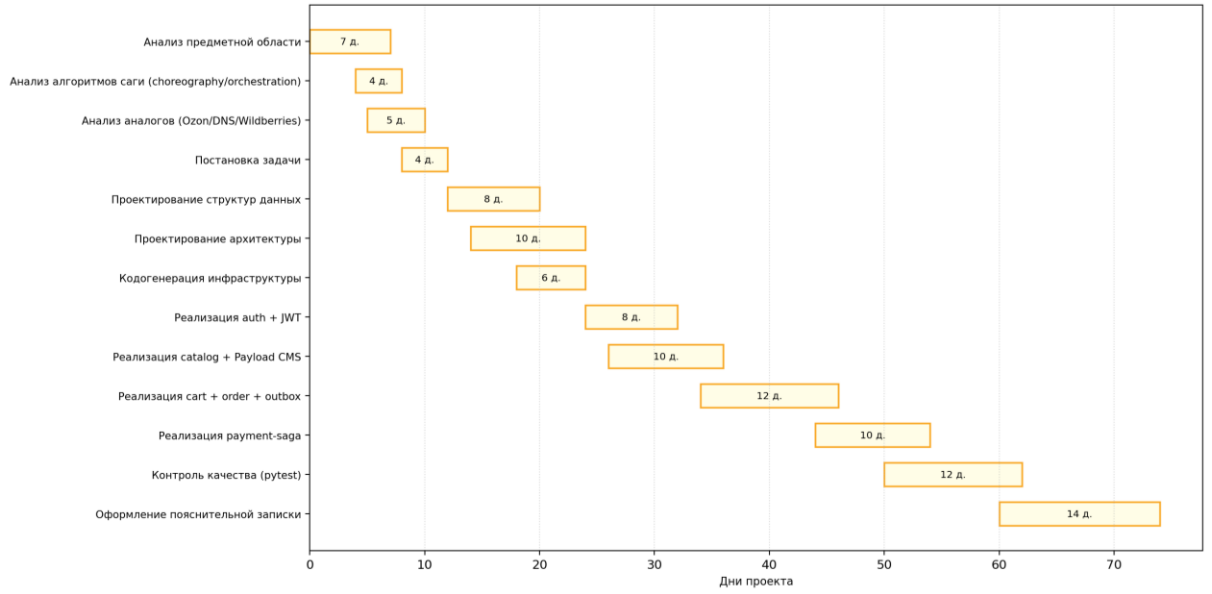


Рисунок 5 – Диаграмма Ганта выполнения работы

В ходе планирования работы над проектом выделено 13 ключевых задач. Началом работы выбрана дата 8 апреля 2025 года; окончание – 13 июня 2025 года. Детализированный список задач представлен на рисунке 6.

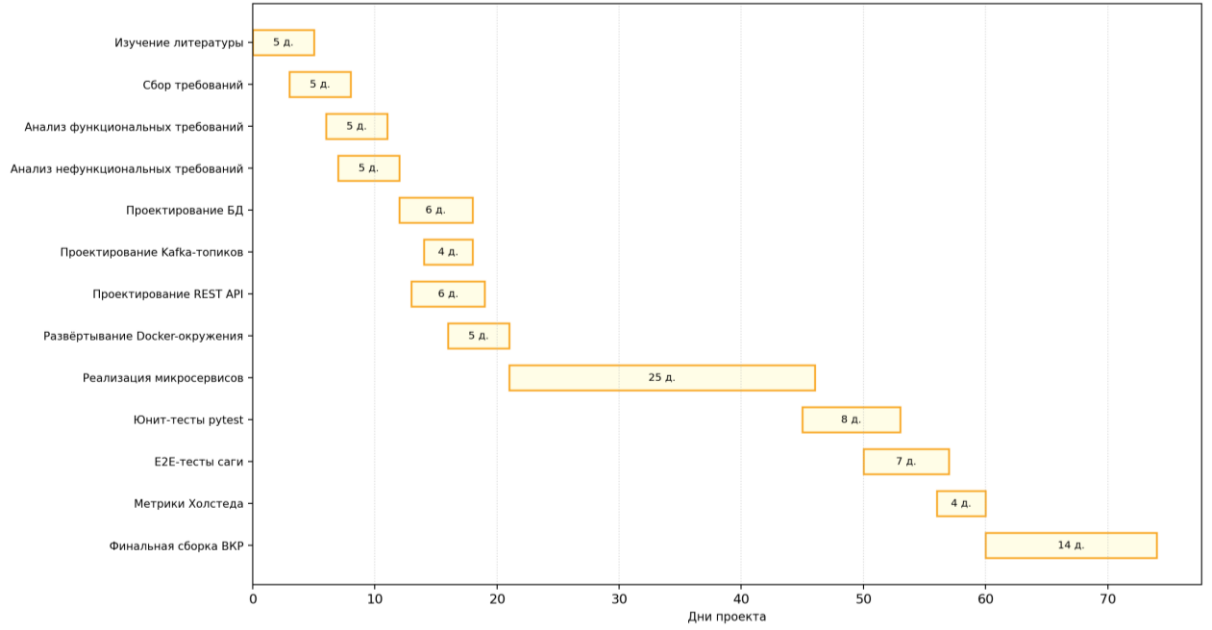


Рисунок 6 – Список задач (Project Libre)

Используемые ресурсы – один разработчик (студент бакалавриата), задействованный на всех этапах разработки. По итогу временные затраты на серверную часть интернет-магазина составили 432 часа, а полная стоимость проекта – 215 000 рублей при ставке 500 руб/час.

2 Проектирование серверной части интернет-магазина

2.1 Архитектура приложения

На основе комплексного изучения функциональных и нефункциональных спецификаций было принято решение о проектировании серверной части онлайн-площадки с использованием микросервисного подхода. Данная архитектурная модель распределенных систем обеспечивает баланс между трудоемкостью разработки и гибкостью внедрения технологий: каждый отдельный модуль может базироваться на стеке, максимально соответствующем его специфике, и взаимодействовать с наиболее эффективным типом СУБД [4].

Для визуализации физической структуры программного комплекса была подготовлена диаграмма компонентов. Она отражает ключевые программные блоки, их взаимосвязи и используемые хранилища. Применение такого инструмента позволяет спроектировать каркас системы, установив корректную логику взаимодействия между мультязычными сервисами и соответствующими базами данных [6]. Графическое представление архитектуры приведено на рисунке 7.

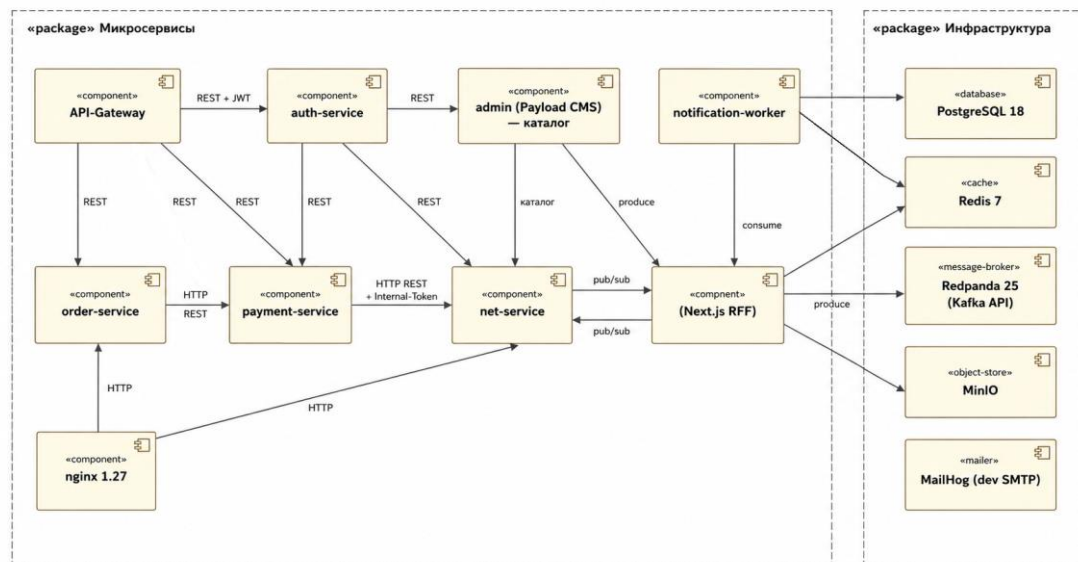


Рисунок 7 – Диаграмма компонентов серверной части

Каждый из изображённых на диаграмме микросервисов отвечает за собственную предметную область:

- API-Gateway – общая точка входа в систему, выполняет JWT-валидацию запросов, прокси-маршрутизацию и трансляцию ошибок;
- auth-service – регистрация, авторизация, ротация refresh-токенов с детектированием угона, восстановление пароля, ролевая модель доступа; при успешной регистрации эмитирует событие `user.registered`, на которое подписан wallet-service для начисления приветственного бонуса;
- order-service – оформление заказа, идемпотентность через Idempotency-Key, transactional outbox для надёжной публикации ``order.created`/`order.cancelled``; подписчик ``payment.confirmed`` для перевода заказа в `paid`;
- payment-service – обработка платежей и хореографическая сага: подписка на ``order.created``, синхронное обращение к wallet-service за списанием средств, публикация ``payment.confirmed`/`payment.failed``;
- wallet-service – внутренний кошелёк пользователя: учёт баланса, идемпотентное списание (debit) и возврат (refund) с защитой через X-Internal-Token, ленивая инициализация при первом обращении к `/wallet/me` с начислением приветственного бонуса 100 000 ₽, публикация ``wallet.debited`/`wallet.refunded``;
- notification-worker – фоновый воркер, подписанный на пять топиков Kafka; рендерит jinja2-шаблоны e-mail и отправляет через SMTP;
- admin (Payload CMS) – встроенная роль каталога и хранилища контента; контент-менеджер ведёт товары и категории через UI, REST-API Payload используется storefront'ом напрямую через server-side proxy.

Для организации взаимодействия между микросервисами применяются два дополняющих подхода: синхронный REST API (через API-Gateway) и асинхронный обмен событиями через брокер Apache Kafka, совместимый с

Redpanda. REST используется для клиентских запросов и операций, требующих немедленного ответа; Kafka – для операций, где допустима эпизодическая согласованность, что снижает связанность сервисов и позволяет точно масштабировать отдельные части системы [7].

Дополнительно в архитектуру включена in-memory СУБД Redis 7, выполняющая три роли: хранение состояния персональных корзин пользователей; SETNX-проверки ключей идиempотентности при создании заказов и обработке внутренних вызовов wallet-service; хранение refresh-токенов с возможностью их отзыва и детектирования повторного использования (token theft detection).

Каталог товаров не выделен в отдельный микросервис. Вместо этого роль single source of truth для каталога выполняет встраиваемая система управления контентом Payload CMS, которая использует ту же PostgreSQL для хранения товаров, категорий и записей о медиа. Storefront получает данные каталога напрямую через server-side проху к Payload REST API. Такое решение позволяет обойтись без отдельного read-side-микросервиса и упрощает схему развёртывания, что отражено в ADR-0016 «Drop Mongo + Meilisearch + catalog-service».

2.2 Проектирование структур данных

2.2.1 Концептуальная модель баз данных

Для описания информационного пространства и семантических связей предметной области, без привязки к техническим особенностям реализации, используется концептуальная модель. Она служит инструментом для формирования единого взгляда на состав данных, подлежащих интеграции в систему [9]. Благодаря высокому уровню абстракции, такая модель позволяет еще на старте проектирования выделить центральные сущности, их свойства и логические зависимости. Опираясь на специфику серверной части онлайн-

маркета, была подготовлена ER-диаграмма («сущность-связь»), приведенная на рисунке 8.

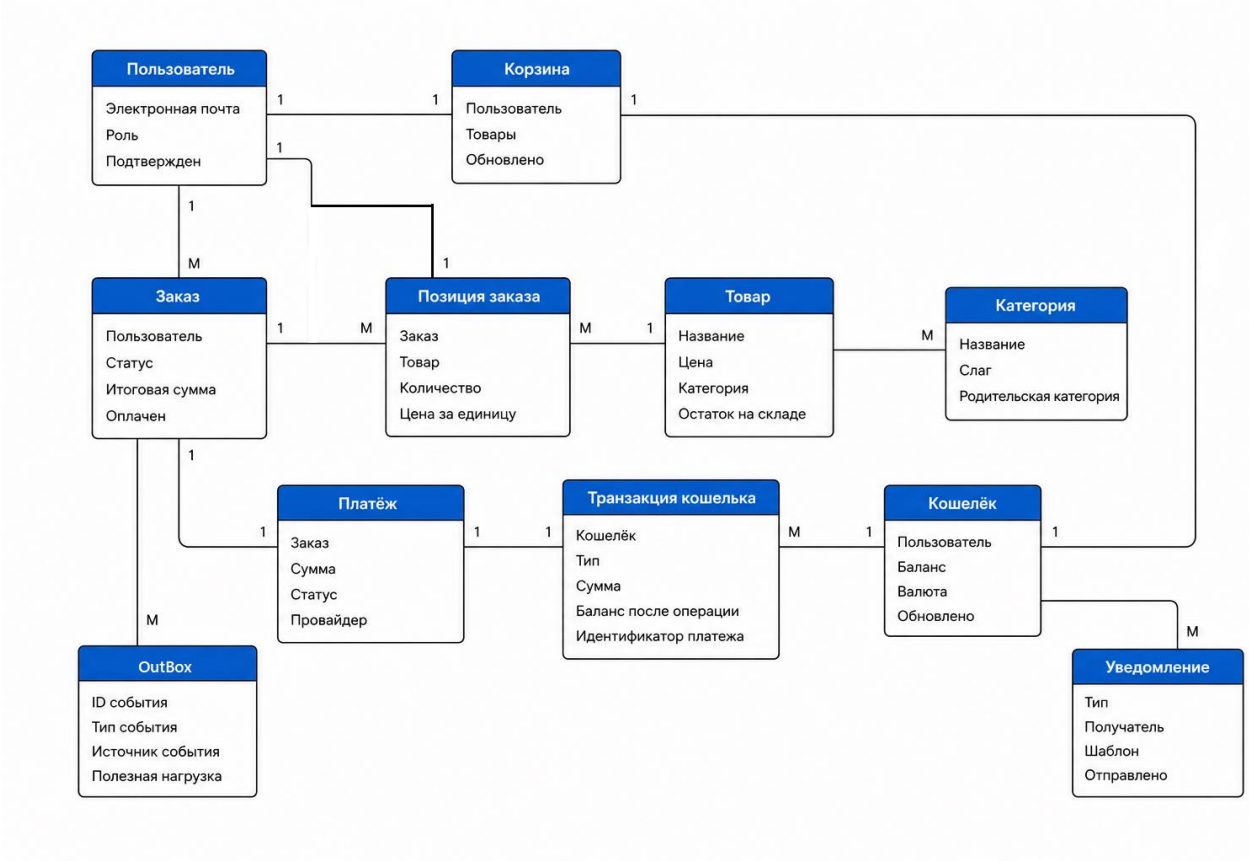


Рисунок 8 – Концептуальная модель базы данных

В таблице 8 представлено описание сущностей, выделенных для организации работы бизнес-процессов интернет-магазина.

Таблица 8 – Описание сущностей

Сущность	Идентификатор	Назначение
Пользователь	user	Учётная запись клиента, контент-менеджера или администратора
Категория	category	Иерархический классификатор товаров
Товар	product	Карточка товара в каталоге
Корзина	cart	Текущий список товаров пользователя до оформления
Заказ	order	Оформленный заказ с историей состояний
Позиция заказа	order_item	Снимок товара в составе заказа

Продолжение таблицы 8

Сущность	Идентификатор	Назначение
Платёж	payment	Запись платёжной операции с указанием provider (wallet/card/sbp)
Кошелёк	wallet	Внутренний баланс пользователя в денежном выражении
Транзакция кошелька	wallet_transaction	Запись об изменении баланса (debit/refund/signup_bonus)
Outbox	outbox	Очередь исходящих CloudEvents до публикации в Kafka
Уведомление	notification_event	История отправленных писем клиентам

В таблице 9 описаны все возможные связи между выделенными сущностями.

Таблица 9 – Описание связей между сущностями

Связь	Описание	Мощность
user-cart	Принадлежность корзины пользователю	1:1
user-order	Принадлежность заказа пользователю	1:M
user-wallet	Принадлежность кошелька пользователю	1:1
wallet-wallet_transaction	Транзакции по кошельку	1:M
order-order_item	Состав заказа	1:M
order_item-product	Снимок товара в позиции заказа	M:1
product-category	Принадлежность товара категории	M:1
order-payment	Платёж, связанный с заказом	1:1
payment-wallet_transaction	Транзакция списания/возврата по платежу	1:1
order-outbox	Исходящие события заказа	1:M
user-notification_event	История уведомлений	1:M

Концептуальная модель разработана в нотации IDEF1X [9] средствами Visual Paradigm 17.

2.2.2 Логическая модель баз данных

Логическая модель данных – расширение концептуальной модели, включающее все сущности, атрибуты, ключи и взаимосвязи, отражающие бизнес-правила [10]. Логическая модель служит основой для физической модели базы данных.

В соответствии с принципом «один микросервис – одна база данных» выделено пять логических баз PostgreSQL: «voltaria_auth», «voltaria_order», «voltaria_payment», «voltaria_wallet» – а также «voltaria_payload» для встраиваемой CMS-каталога. Такое разделение мотивировано следующим: независимое управление по предметной области; снижение нагрузки на любую отдельную БД; возможность независимого масштабирования и резервного копирования. На рисунке 9 представлена логическая модель БД микросервиса авторизации.

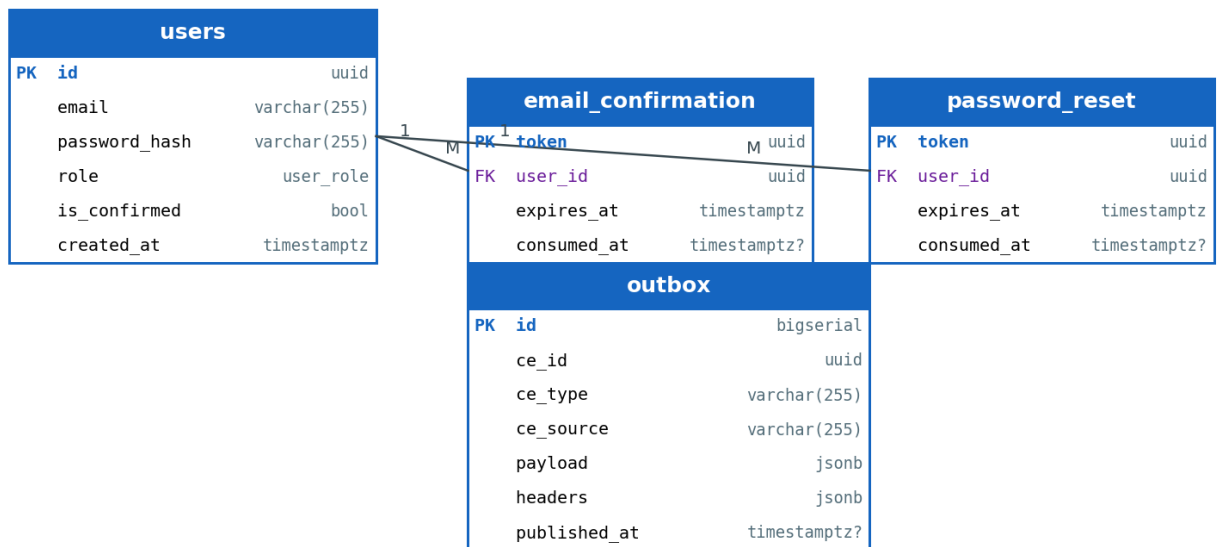


Рисунок 9 – Логическая модель БД микросервиса авторизации

База данных «voltaria_auth» выделена в отдельный контур для обеспечения автономной работы и безопасности сервиса авторизации.

Представленная логическая модель позволяет независимо масштабировать хранилище и оптимизировать обработку учетных данных. На рисунке 10 представлена логическая модель БД микросервиса заказов.

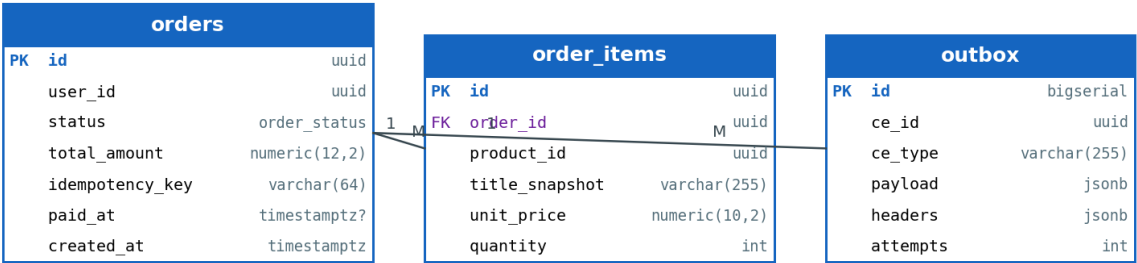


Рисунок 10 – Логическая модель БД микросервиса заказов

Изоляция БД «voltaria_order» обеспечивает полную автономность процессов обработки заказов и исключает зависимость от состояния других компонентов системы. Такое решение позволяет оптимизировать хранилище под высокую транзакционную нагрузку и гарантирует целостность данных на всех этапах жизненного цикла заказа. На рисунке 11 изображена логическая модель БД микросервиса платежей.

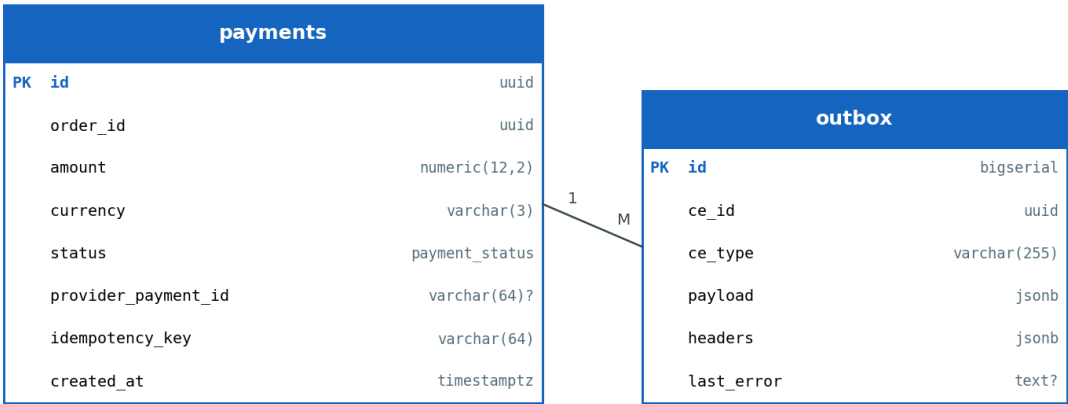


Рисунок 11 – Логическая модель БД микросервиса платежей

Изоляция базы данных «voltaria_payment» обеспечивает высокую безопасность и отказоустойчивость при обработке финансовых транзакций, полностью исключая влияние соседних модулей на платежные операции. Это позволяет гибко настраивать производительность хранилища под специфические нагрузки и гарантирует сохранность критически важных данных в независимом контуре. На рисунке 12 представлена логическая модель БД микросервиса внутреннего кошелька.

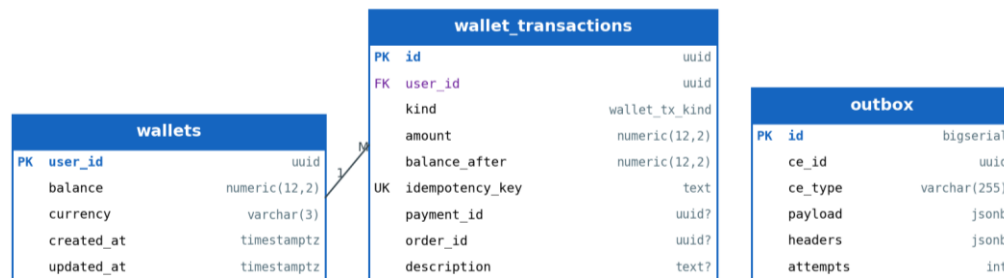


Рисунок 12 – Логическая модель БД микросервиса внутреннего кошелька

Выделение базы данных «voltaria_wallet» в отдельный сегмент гарантирует максимальную сохранность финансовых балансов и исключает влияние других сервисов на операции со счетами. На рисунке 13 представлена логическая модель БД микросервиса каталога (Payload CMS).

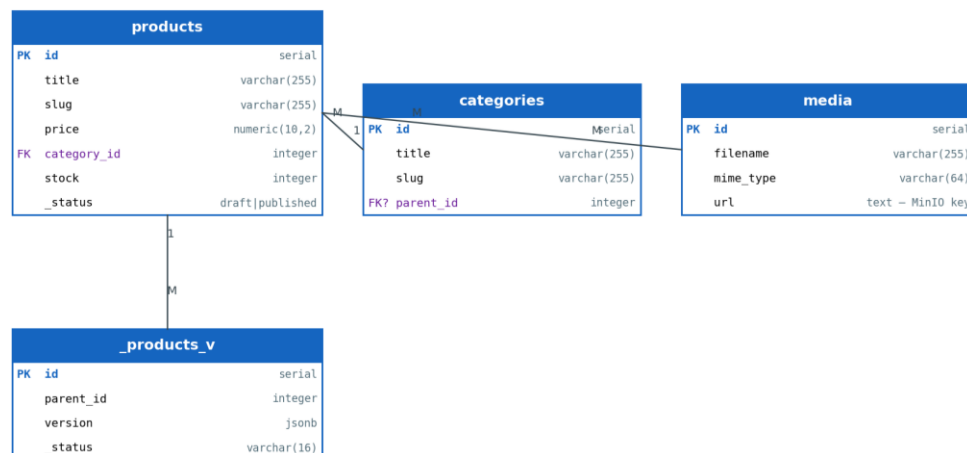


Рисунок 13 – Логическая модель БД микросервиса каталога (Payload CMS)

Разработанные логические модели в дальнейшем используются при формировании физической модели и Alembic-миграций.

2.3 Обобщённая структура разрабатываемого сервиса

В общем виде разрабатываемый интернет-магазин представляет собой клиент-серверное приложение. Клиентская часть – браузер пользователя, в котором исполняется одностраничное приложение storefront, построенное на Next.js 16 [19]. Сервер-приложение запускается на удалённой машине под управлением Docker и состоит из шести микросервисов, обратного прокси-сервера nginx, объектного хранилища MinIO, брокера Redpanda [20] и внутреннего кеша Redis. Хранение реляционных данных выполняется на отдельном сервере БД с PostgreSQL 18.

Основные преимущества такого построения:

- отсутствие дублирования: все микросервисы, статические ресурсы и БД находятся в одной управляемой инфраструктуре;
- минимальные требования к пользователю – достаточно браузера;
- безопасность: данные хранятся в БД, пользователи не получают к ним прямого доступа.

Основные недостатки клиент-серверного построения:

- выход из строя сервера или БД делает сервис недоступным для всех клиентов;
- возможны DDoS-атаки, замедляющие отклик [11].

Использование контейнеризации и микросервисного подхода позволяет гибко управлять ресурсами системы и оперативно внедрять новые функциональные возможности без остановки всего комплекса. Предложенная конфигурация серверных компонентов и связей между ними формирует устойчивую среду, способную эффективно обрабатывать запросы и распределять нагрузку. Приведенная далее диаграмма развертывания наглядно

демонстрирует физическое размещение программных модулей и их сетевое взаимодействие в рамках единой инфраструктуры проекта. На рисунке 14 приведена диаграмма развёртывания.

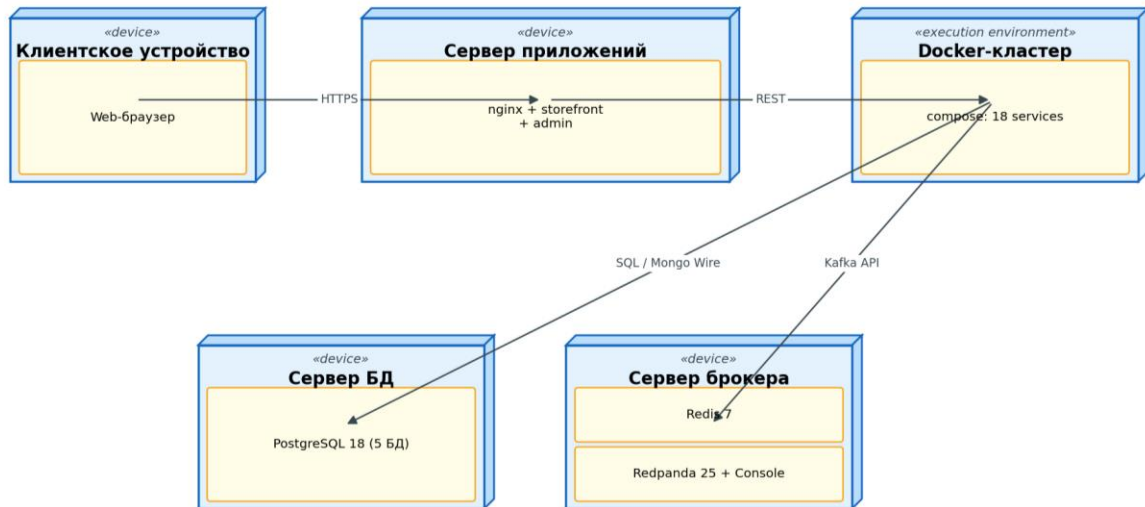


Рисунок 14 – Диаграмма развёртывания

Пользователь отправляет HTTP-запросы из браузера в nginx, который маршрутизирует их между storefront и admin (Payload CMS). Серверные приложения, в свою очередь, обращаются к серверу БД по SQL-протоколу PostgreSQL, а к серверу брокера – по Kafka API. Внутренние обращения payment-service → wallet-service выполняются поверх REST с обязательным заголовком X-Internal-Token. Для развёртывания на обоих серверах должен быть установлен Docker.

2.4 Требования к аппаратному обеспечению серверной части

Для корректной работы серверной части интернет-магазина выделены минимальные и рекомендуемые требования к техническому оснащению. Эти требования описаны в таблице 10. Указанные параметры рассчитаны исходя из суммарной нагрузки всех микросервисов и инфраструктурных компонентов в режиме пиковой активности. Соблюдение данных спецификаций критически

важно для обеспечения бесперебойного функционирования системы и поддержания высокой скорости отклика при обработке пользовательских запросов.

Таблица 10 – Требования к программному и аппаратному обеспечению

Характеристика	Минимальные требования	Рекомендуемые требования
Процессор	x64, 4 ядра, 2,0 ГГц	x64, 8 ядер, 3,0 ГГц и выше
Оперативная память	8 Гб	16 Гб и выше
Тип и объём диска	SSD 100 Гб	NVMe SSD 1 Тб и выше
Пропускная способность сети	100 Мбит/с	1 Гбит/с
Операционная система	Ubuntu 22.04 LTS	Ubuntu 24.04 LTS
Docker Engine	24.0	27.0 и выше
PostgreSQL	16.0	18.0 и выше

Минимальные требования позволяют обеспечить базовую функциональность системы при нагрузке до 50 одновременных пользователей. Рекомендуемые параметры направлены на достижение оптимальной производительности при пиковых нагрузках до 500 одновременных пользователей и поддержание стабильного отклика API.

3 Разработка программного обеспечения серверной части

3.1 Разработка архитектуры серверной части

Бэкенд рассматриваемой e-commerce платформы базируется на принципах микросервисного проектирования. Согласно определению [12], под микросервисами понимаются автономные программные единицы, сгруппированные по функциональным областям бизнеса. Такие компоненты взаимодействуют через сетевые протоколы, обеспечивая полную изоляцию своей внутренней логики и механизмов хранения данных от внешней среды.

Архитектура системы включает шесть специализированных сервисов, каждый из которых управляет конкретным бизнес-доменом и предоставляет собственный REST-интерфейс. Для обеспечения независимости данных каждый микросервис работает с персональной базой данных через паттерн «Репозиторий». Взаимодействие фронтенда с серверной частью организовано через шлюз API Gateway, выполняющий роль единой точки входа. Его задачи включают валидацию JWT-токенов, маршрутизацию запросов к нужным узлам и унификацию ответов об ошибках. Примечательно, что для взаимодействия между платежным сервисом (payment-service) и кошельком (wallet-service) используется прямой вызов, защищенный специальным заголовком X-Internal-Token, значение которого извлекается из переменных окружения.

Внутренняя структура каждого модуля опирается на многоуровневый шаблон проектирования. Данный подход предполагает разделение функционала на изолированные слои, что, согласно [13], существенно упрощает сопровождение системы, повышает модульность и позволяет повторно использовать программные компоненты. Такая организация архитектуры способствует масштабируемости системы, позволяя при необходимости независимо развивать и обновлять отдельные сервисы без влияния на остальные компоненты. Это также повышает устойчивость к сбоям, поскольку отказ

одного модуля не приводит к полной деградации всей системы. Схематично это разделение продемонстрировано на рисунке 15.

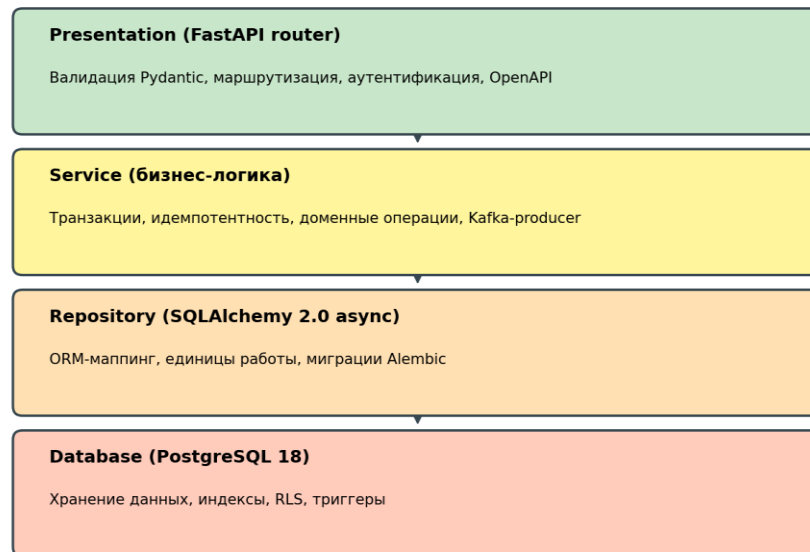


Рисунок 15 – Многослойная архитектура микросервиса

Архитектурные слои обеспечивают поэтапную обработку входящих транзакций. Процесс начинается на уровне представления (контроллеры FastAPI), где осуществляется первичная валидация данных с применением схем Pydantic. После проверки управление передается в сервисный слой (Service), в котором сосредоточена ключевая бизнес-логика и операции над доменными сущностями. Для взаимодействия с персистентным хранилищем сервисы используют слой репозитория (Repository). Данный уровень абстракции реализован на базе SQLAlchemy 2.0 в асинхронном режиме и отвечает за трансляцию ORM-моделей в запросы к СУБД. Сама база данных не только аккумулирует информацию, но и содержит вспомогательные механизмы: индексы, триггерные функции и средства агрегации.

Взаимодействие компонентов системы выстроено согласно парадигме REST API, что гарантирует унификацию HTTP-интерфейсов. Для реализации событийной модели и выполнения фоновых задач интегрирован брокер

сообщений Apache Kafka (в реализации Redpanda). Механизм обмена данными основан на паттерне «издатель-подписчик»: компонент-источник (producer) транслирует типизированное сообщение в соответствующий топик, после чего оно поступает на обработку конечному потребителю (consumer).

Все события упакованы в конверт CloudEvents 1.0. Атрибуты ``specversion``, ``id``, ``source`` и ``type`` дублируются в Kafka-заголовках ``ce_specversion`/`ce_id`/`ce_source`/`ce_type``, что позволяет фильтровать события без десериализации тела сообщения. Регистрация типов событий централизована в shared-пакете ``voltaria_shared.events.registry``, что делает невозможным отправку нетипизированного события.

Для снятия нагрузки с основной БД и временного хранения данных используется key-value-СУБД Redis. Её применение позволяет меньше нагружать PostgreSQL, кешируя выдачу каталога и удерживая корзину пользователя. Поскольку Redis хранит данные в оперативной памяти, выборка из неё не требует обращения к другим микросервисам – только к локальному кэшу.

Для надёжной публикации событий применён шаблон transactional outbox. В каждой БД микросервиса присутствует таблица ``outbox`` с обязательными колонками ``ce_id``, ``ce_type``, ``ce_source``, ``payload``, ``headers``, ``attempts``. Бизнес-операция и вставка строки в ``outbox`` выполняются в одной транзакции SQLAlchemy. Отдельный фоновый процесс (``relay_outbox_loop``) внутри сервиса периодически выбирает неопубликованные записи и публикует их в Kafka, пометчая ``published_at``. Это гарантирует, что событие никогда не будет потеряно даже при сбое до публикации.

Хореографическая сага оформления и оплаты заказа с внутреннего кошелька, построенная на основе этих компонентов, представлена на рисунке 16 – последовательность сообщений между Storefront BFF, order-service, Kafka, payment-service и wallet-service.

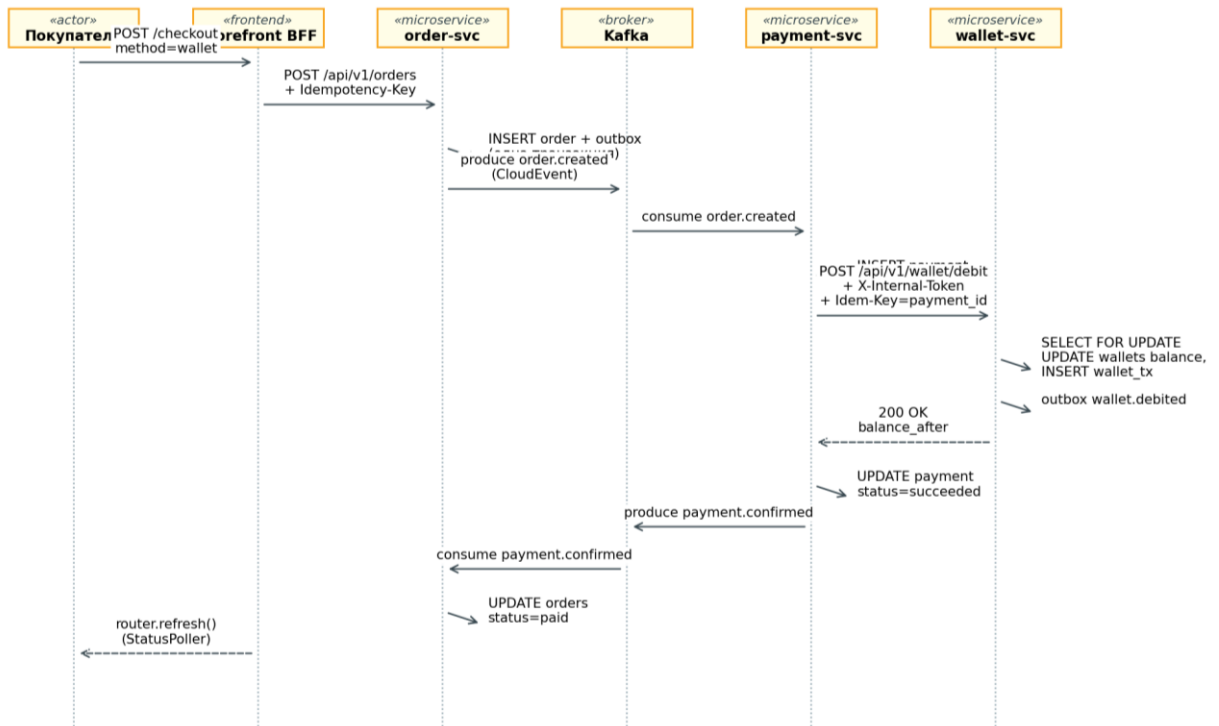


Рисунок 16 – Хореографическая сага оплаты заказа с кошелька

Диаграмма классов основных компонентов саги и кошелька представлена на рисунке 17.

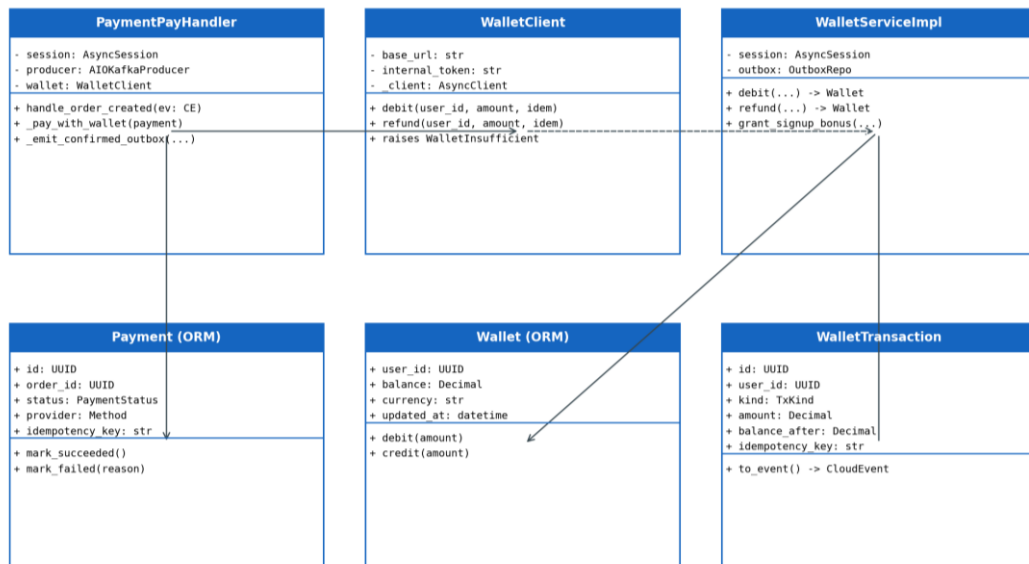


Рисунок 17 – Диаграмма классов модулей Payment-Saga и Wallet

Помимо записи и обработки команд, серверная часть реализует CQRS read-projection временной линии саги. Order-service содержит фоновый consumer, подписанный на пять Kafka-топиков (`order.created`, `order.cancelled`, `payment.requested`, `payment.confirmed`/`payment.failed`, `wallet.debited`/`wallet.refunded`). Каждое событие сохраняется в таблице `order\_events` со ссылкой на `order\_id`, типом и временем; одновременно `orders.status` обновляется в соответствии с состоянием саги. Эндпоинт `GET/api/v1/orders/{id}/events` возвращает упорядоченный список событий, отображаемый клиентом как live-таймлайн (рисунок 18). Это даёт пользователю наглядное представление о промежуточных шагах асинхронной операции и одновременно является основой для аудита и наблюдаемости системы.

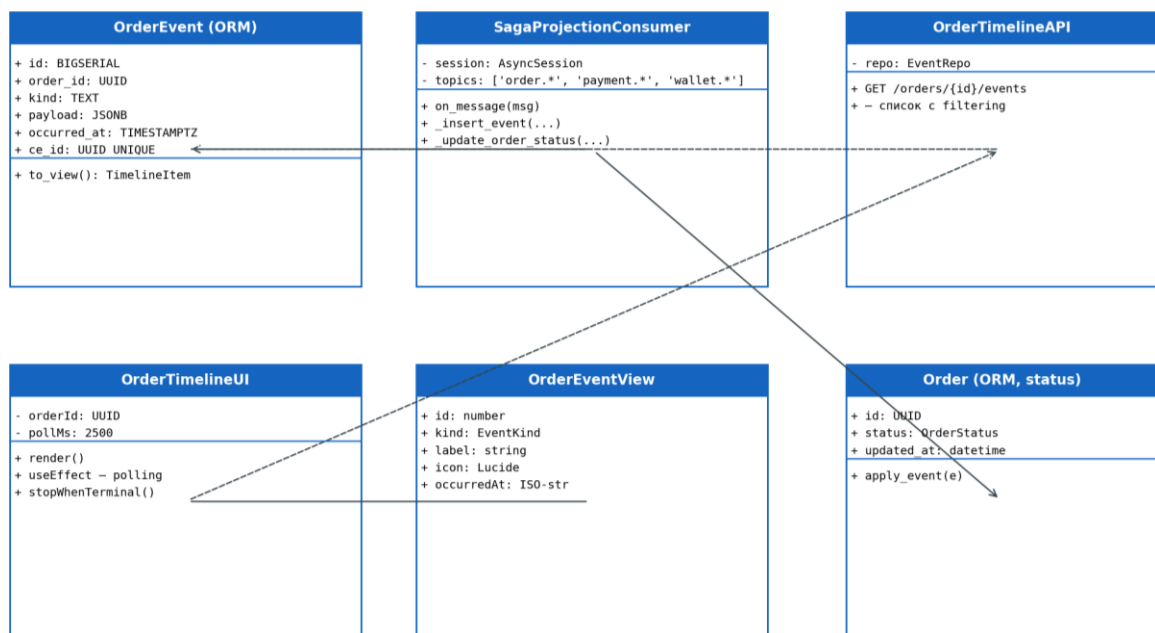


Рисунок 18 – CQRS read-projection «Order Saga Timeline»

Таким образом, реализация CQRS read-projection позволяет трансформировать сложные асинхронные переходы саги в прозрачный и понятный поток данных. Это не только улучшает пользовательский опыт, предоставляя актуальный статус заказа, но и обеспечивает систему надежным

механизмом аудита, необходимым для диагностики распределенных транзакций.

3.2 Разработка физической модели базы данных

Физическая модель базы данных – финальный этап моделирования, выполненный на основе логической модели. Эта модель включает все необходимые таблицы, столбцы, связи и свойства БД для физической реализации. Производительность, стратегия индексации, физическое хранилище и денормализация – важные параметры физической модели [14]. Физическая модель разрабатывается под конкретную СУБД (PostgreSQL 18).

В ходе выполнения работы были разработаны физические модели четырёх ключевых баз данных: `voltaria\_auth`, `voltaria\_order`, `voltaria\_payment` и `voltaria\_wallet`. Физическая модель каталога управляется автоматически Payload CMS через миграции drizzle-orm и приведена в приложении К. На рисунке 19 представлена физическая модель БД «voltaria_auth».

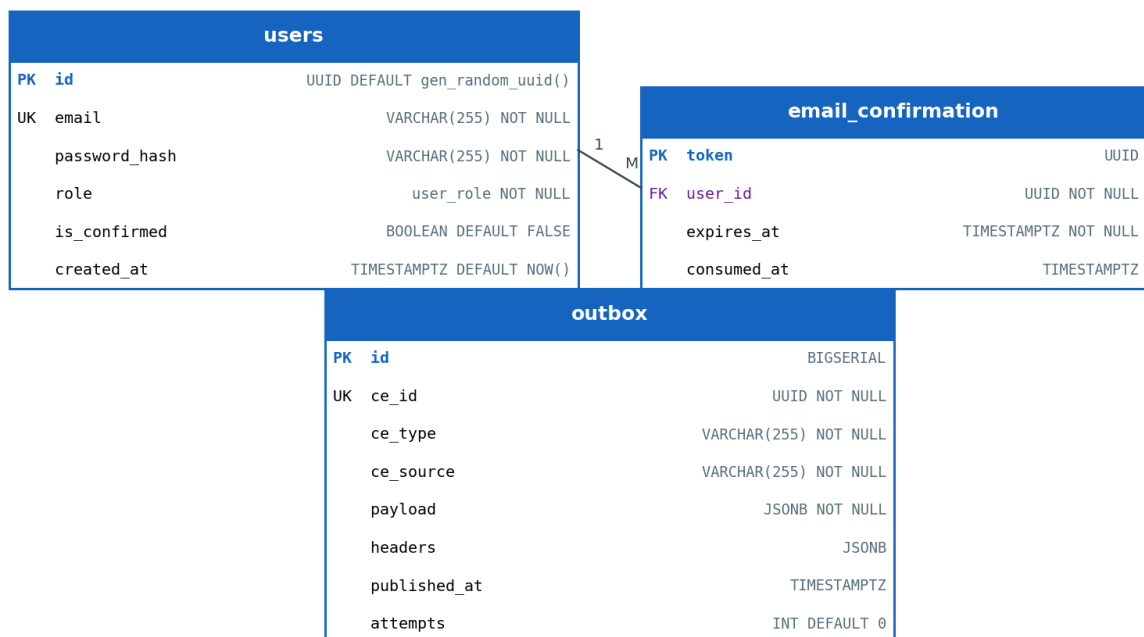


Рисунок 19 – Физическая модель БД «voltaria_auth»

Физическая модель базы данных «voltaria_auth» описывает детальную структуру хранения данных, включая спецификации типов полей, индексы и ограничения целостности. Выбранная конфигурация таблиц и связей оптимизирована для обеспечения минимальных задержек при выполнении процедур аутентификации и высокой надежности хранения учетных записей. На рисунке 20 представлена физическая модель БД «voltaria_order».

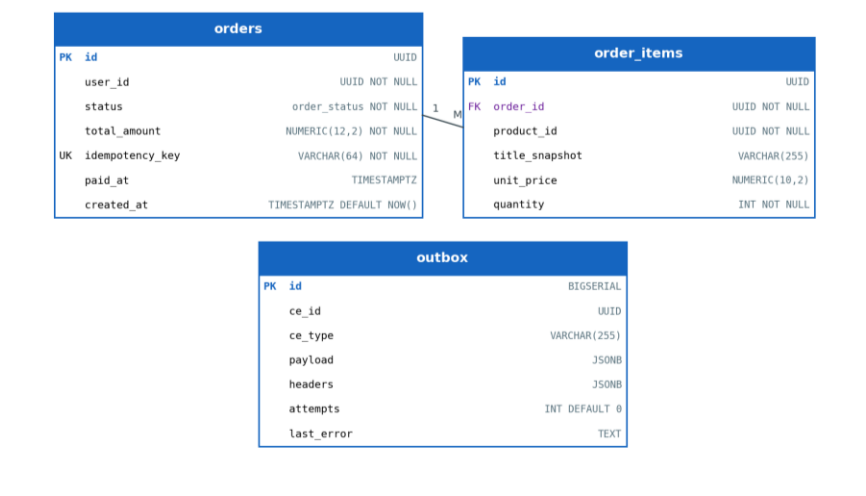


Рисунок 20 – Физическая модель БД «voltaria_order»

Физическая модель БД «voltaria_order» детально определяет структуру таблиц и индексов, необходимых для эффективного хранения данных о заказах и их статусах. На рисунке 21 представлена физическая модель БД «voltaria_payment».



Рисунок 21 – Физическая модель БД «voltaria_payment»

Физическая модель БД «voltaria_payment» регламентирует техническую реализацию хранения данных о транзакциях и платежных операциях. Оптимизированная структура таблиц и индексов обеспечивает высокую скорость записи и целостность финансовых записей, что критически важно для корректной работы платежного шлюза. На рисунке 22 представлена физическая модель БД «voltaria_wallet».

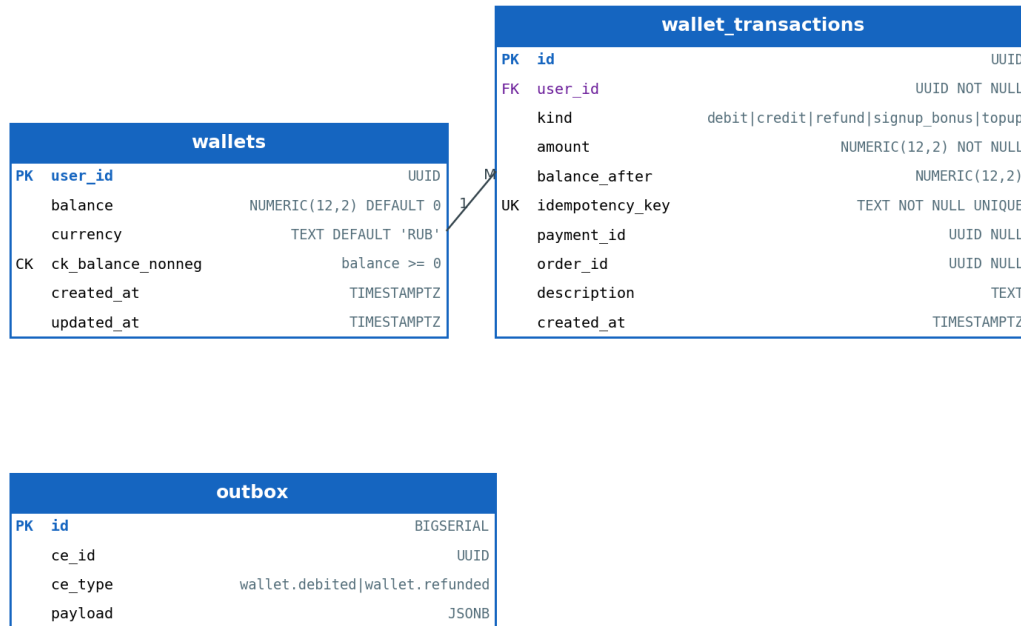


Рисунок 22 – Физическая модель БД «voltaria_wallet»

Особенностью физической модели является обязательное использование типа `'TIMESTAMP WITH TIME ZONE'` для всех временных колонок. Чисто аннотационный `'Mapped[datetime]'` в SQLAlchemy 2.0 транслируется в `'TIMESTAMP WITHOUT TIME ZONE'`, что приводит к ошибкам asynpcpg «can't subtract offset-naive and offset-aware datetimes» при INSERT с tz-aware-значениями. Поэтому в каждом микросервисе все datetime-колонки явно объявлены как `'mapped_column(TIMESTAMP(timezone=True), ...)'`.

В таблице 11 представлено описание сущностей всех баз данных серверной части.

Таблица 11 – Описание сущностей баз данных

№	Сущность	Описание	Категория
1	users	Учётные записи всех пользователей	Стержневая
2	email_confirmation	Токены подтверждения e-mail	Характеристическая
3	password_reset	Токены восстановления пароля	Характеристическая
4	orders	Оформленные заказы покупателей	Стержневая
5	order_items	Позиции в составе заказа	Характеристическая
6	payments	Платёжные операции по заказам с указанием provider	Стержневая
7	wallets	Внутренние кошельки пользователей с балансом	Стержневая
8	wallet_transactions	История изменений баланса (debit/refund/signup_bonus/topup)	Характеристическая
9	outbox	Очередь исходящих CloudEvents	Стержневая
10	products (Payload)	Каталог товаров (Payload CMS)	Стержневая
11	categories (Payload)	Иерархия категорий каталога	Кодовая
12	products (Meili)	Поисковый индекс товаров	Кодовая

Спецификации атрибутов всех сущностей таблиц БД представлены в приложении И. Структуры Pydantic-моделей для слоёв обращения к БД описаны в приложении Б. SQL-скрипты для создания и наполнения таблиц представлены в приложении В.

3.3 Разработка архитектурных слоёв внутри микросервисов

В микросервисной архитектуре критически важно чёткое разделение ответственности между компонентами системы. Для обеспечения масштабируемости, тестируемости и поддерживаемости кода каждый микросервис построен по принципу многослойной архитектуры.

В качестве примера детального рассмотрения структуры и взаимодействия слоёв будет использоваться микросервис обработки платежей

(payment-service). Этот выбор обусловлен тем, что в нём реализованы все ключевые архитектурные шаблоны проекта: хореографическая сага, идемпотентность через SETNX, transactional outbox, обработка веб-хуков платёжного шлюза с гарантией дедупликации. Реализация остальных микросервисов выполнена аналогично; их детальное описание приведено в приложениях Г, Д, Е.

3.3.1 Слой представления данных

Слой представления данных описывает контракт, по которому клиент узнаёт, какие операции доступны в микросервисе. В payment-service слой реализован через FastAPI APIRouter. Каждый эндпоинт декларирует Pydantic-модели запроса и ответа и подключается к общему ASGI-приложению при инициализации. В листинге 1 представлена регистрация APIRouter в FastAPI-приложении.

Листинг 1 – Регистрация APIRouter в FastAPI-приложении

```
from fastapi import FastAPI
from .routes import payments_router, webhook_router, health_router

def create_app(*, with_lifespan: bool = True) -> FastAPI:
    app = FastAPI(title="payment-service", lifespan=lifespan if
with_lifespan else None)
    app.include_router(health_router)
    app.include_router(payments_router, prefix="/api/v1/payments")
    app.include_router(webhook_router, prefix="/api/v1/payments")
    return app
```

Особенностью эндпоинта приёма веб-хука является отсутствие требования авторизации (заголовок Authorization не передаётся платёжным шлюзом) и обязательная проверка Idempotency-Key. В листинге 2 приведена сигнатура соответствующего обработчика.

Листинг 2 – Обработчик POST /payments/webhook

```
@webhook_router.post("/webhook", status_code=status.HTTP_200_OK)
```

```

async def webhook(
    payload: WebhookPayloadIn,
    idempotency_key: str = Header(..., alias="Idempotency-Key"),
    service: PaymentWebhookService = Depends(get_webhook_service),
) -> WebhookResponseOut:
    payment = await service.process_webhook(
        payment_id=payload.payment_id,
        new_status=payload.status,
        idempotency_key=idempotency_key,
    )
    return WebhookResponseOut(payment_id=payment.id,
status=payment.status)

```

Представленный листинг демонстрирует реализацию обработчика вебхуков для платежной системы. Он обеспечивает асинхронную обработку уведомлений о статусах платежей, используя ключ идемпотентности для предотвращения повторных операций. Логика обновления статуса делегирована отдельному сервису, что повышает модульность и тестируемость кода.

3.3.2 Слой бизнес-логики

Слой бизнес-логики реализует основной алгоритм микросервиса и взаимодействует с репозиторием, Redis, Kafka-продюсером и внешним платёжным шлюзом. Ключевой метод – `process\_webhook` – реализует логику перевода платежа в финальное состояние с гарантией дедупликации. В листинге 3 представлена реализация метода `process_webhook` сервисного слоя.

Листинг 3 – Метод `process_webhook` сервисного слоя

```

async def process_webhook(self, *, payment_id, new_status,
idempotency_key):
    if not await self.redis.setnx(f"webhook:{idempotency_key}",
"1"):
        return await self.repo.get_by_id(payment_id)
    await self.redis.expire(f"webhook:{idempotency_key}", 86400)

    async with self.session.begin():
        payment = await self.repo.get_for_update(payment_id)
        if payment.status in TERMINAL_STATES:
            return payment
        payment.status = new_status
        payment.confirmed_at = utcnow()

```

```

        await self.outbox.append(
            ce_type="payment.confirmed" if new_status ==
"succeeded" else "payment.failed",
            payload=payment.to_event(),
        )
    return payment

```

Метод реализует все требования к идемпотентному обработчику веб-хука: проверка дедупликации в Redis через SETNX; пессимистичная блокировка строки платежа через SELECT FOR UPDATE; проверка терминальных состояний; запись CloudEvent в outbox в той же транзакции, что и UPDATE.

3.3.3 Слой обращения к базе данных

Слой обращения к БД построен на SQLAlchemy 2.0 в асинхронном режиме (async/await) [16]. ORM-модели описываются через `Mapped[T]` с явными типами колонок. В листинге 4 описана ORM-модель Payment.

Листинг 4 – ORM-модель Payment

```

class Payment(Base):
    __tablename__ = "payments"
    id: Mapped[UUID] = mapped_column(primary_key=True,
default=uuid4)
    order_id: Mapped[UUID] = mapped_column(nullable=False)
    amount: Mapped[Decimal] = mapped_column(Numeric(12, 2),
nullable=False)
    currency: Mapped[str] = mapped_column(String(3), default="RUB")
    status: Mapped[PaymentStatus] = mapped_column(
        SQLAEnum(PaymentStatus, name="payment_status"),
        nullable=False
    )
    idempotency_key: Mapped[str] = mapped_column(String(64),
unique=True)
    provider_payment_id: Mapped[str | None] =
mapped_column(String(64))
    created_at: Mapped[datetime] = mapped_column(
        TIMESTAMP(timezone=True), server_default=func.now()
    )
    confirmed_at: Mapped[datetime | None] =
mapped_column(TIMESTAMP(timezone=True))

```

Репозиторий инкапсулирует CRUD-операции и специфические запросы вроде `get\_for\_update`, выполняющие пессимистичную блокировку. В листинге 5 представлен метод `get_for_update` репозитория.

Листинг 5 – Метод `get_for_update` репозитория

```
async def get_for_update(self, payment_id: UUID) -> Payment:
    stmt = select(Payment).where(Payment.id ==
payment_id).with_for_update()
    result = await self.session.execute(stmt)
    payment = result.scalar_one_or_none()
    if payment is None:
        raise PaymentNotFoundError(payment_id)
    return payment
```

Схема БД управляется Alembic-миграциями, развёртываемыми автоматически при старте контейнера через `alembic upgrade head`. Полный листинг репозитория представлен в приложении Е.

4 Контроль качества программного обеспечения

4.1 Обоснование выбора модели тестирования и метрик кода

Контроль качества программного обеспечения – обязательный этап разработки, обеспечивающий соответствие реализации поставленным требованиям и устойчивость к ошибкам. Для серверной части интернет-магазина выбрана пирамидальная модель тестирования: модульные тесты составляют основу пирамиды и покрывают отдельные функции и методы; интеграционные тесты проверяют взаимодействие сервиса с его базой данных, Redis и Kafka через testcontainers; функциональные (E2E) тесты проверяют сквозную сагу через REST API.

В качестве тестового фреймворка использован pytest 8 с pytest-asyncio для поддержки асинхронных тестов и pytest-testcontainers для управления Docker-зависимостями [17]. На текущий момент серверная часть содержит более 130 автоматических тестов, выполняющихся в CI за 35 - 40 секунд.

Для количественной оценки качества кода применены метрики Холстеда [15]. Эти метрики позволяют оценить сложность кода по его словарю и длине, что важно для прогнозирования сложности сопровождения и числа потенциальных ошибок.

4.2 Тестирование серверной части интернет-магазина

4.2.1 Функциональное тестирование саги оплаты заказа

Функциональное тестирование выполняется по схеме «чёрный ящик» – без доступа к внутренним деталям реализации. Тестируется сквозной бизнес-сценарий «оформить и оплатить заказ», затрагивающий три микросервиса (order-service, payment-service, notification-worker), Kafka, два Postgres-кластера и Redis [18]. В таблице 12 представлены тест-кейсы функционального тестирования.

Таблица 12 – Тест-кейсы функционального тестирования

№	Сценарий	Ожидаемый результат	Статус
ТС-01	Регистрация нового пользователя	user.registered → wallet-service создаёт wallet, начисляет signup_bonus 500 Р	PASS
ТС-02	Создание заказа с уникальным Idempotency-Key	201 Created. Order в БД. Outbox содержит `order.created`	PASS
ТС-03	Повторный POST с тем же Idempotency-Key	201 Created. Тот же order_id. Outbox без дубликата	PASS
ТС-04	Оплата заказа из кошелька (баланс достаточен)	Order.status=paid в течение 1500 мс. Wallet.balance уменьшен. Outbox `payment.confirmed` + `wallet.debited`	PASS
ТС-05	Оплата заказа из кошелька (недостаточно средств)	402 от wallet-service. Payment.status=failed. Order.status=cancelled. Outbox `payment.failed`	PASS
ТС-06	Повторный debit с тем же Idempotency-Key=payment_id	200 OK. Wallet.balance без изменений (idempotency)	PASS
ТС-07	Отмена заказа после оплаты – refund на кошелёк	Wallet.balance увеличивается. WalletTx(kind='refund'). Outbox `wallet.refunded`	PASS
ТС-08	Service-to-service вызов wallet/debit без X-Internal-Token	401 Unauthorized	PASS
ТС-09	Сбой Kafka на этапе саги	Outbox содержит событие, после восстановления – публикация	PASS
ТС-10	Параллельные списания (race condition)	Только одна транзакция применяется (SELECT FOR UPDATE), остаток корректен	PASS

Результаты функционального тестирования подтверждают корректность реализации ключевых бизнес-процессов и стабильность межсервисного взаимодействия. Успешное прохождение всех тест-кейсов, включая проверку механизмов идемпотентности, обработки race conditions и отказоустойчивости брокера сообщений, гарантирует высокую надежность системы и целостность данных при выполнении транзакционных операций.

4.2.2 Модульное тестирование обработчика веб-хука

Модульное тестирование выполняется в изоляции от внешних зависимостей – Postgres подменяется in-memory-fixture, Redis – через `fakeredis`, Kafka – мок-объектом `AIOKafkaProducer`. На листинге 6 представлен фрагмент модульного теста идемпотентного повторного приёма веб-хука.

Листинг 6 – Модульный тест process_webhook (выдержка)

```
@pytest.mark.asyncio
async def test_process_webhook_dedup_on_second_call(
    webhook_service: PaymentWebhookService,
    pending_payment: Payment,
) -> None:
    first = await webhook_service.process_webhook(
        payment_id=pending_payment.id,
        new_status=PaymentStatus.SUCCEEDED,
        idempotency_key="abc123",
    )
    second = await webhook_service.process_webhook(
        payment_id=pending_payment.id,
        new_status=PaymentStatus.SUCCEEDED,
        idempotency_key="abc123",
    )
    assert first.id == second.id == pending_payment.id
    outbox = await read_outbox(webhook_service.session)
    assert len([e for e in outbox if e.ce_type ==
"payment.confirmed"]) == 1
```

Тест проверяет ключевое требование к веб-хуку: повторный приём одного и того же сообщения от платёжного шлюза не порождает дубликата CloudEvent. Этим обеспечивается ровно-однократная семантика для подписчиков (order-service, notification-worker).

4.2.3 Метрики Холстеда метода создания заказа

Для метода 'OrderService.create_order' рассчитаны метрики Холстеда. Метод выбран как наиболее ответственный: он реализует идемпотентность через Idempotency-Key, проверяет остатки через Payload, выполняет INSERT+outbox в одной транзакции и обладает повышенным риском ошибок при сопровождении.

Метрики Холстеда [15] основаны на двух первичных параметрах: η_1 – число уникальных операторов, η_2 – число уникальных операндов; и двух вторичных параметрах: N_1 – общее число операторов, N_2 – общее число операндов. На их основе вычисляются производные метрики: словарь $\eta = \eta_1 + \eta_2$; длина программы $N = N_1 + N_2$; объём $V = N \cdot \log_2 \eta$; сложность $D = (\eta_1/2) \cdot (N_2/\eta_2)$; трудоёмкость $E = D \cdot V$; оценка числа ошибок $B = E^{(2/3)} / 3000$. В таблице 13 приведены метрики Холстеда для метода OrderService.create_order.

Таблица 13 – Метрики Холстеда для метода «OrderService.create_order»

Метрика	Значение	Описание
η_1 (операторы)	21	Уникальные ключевые слова, операторы, вызовы
η_2 (операнды)	34	Уникальные имена переменных и литералов
N_1	52	Общее число вхождений операторов
N_2	61	Общее число вхождений операндов
η (словарь)	55	$\eta_1 + \eta_2$
N (длина)	113	$N_1 + N_2$
V (объём)	653,4 бит	$N \cdot \log_2 \eta$
D (сложность)	18,8	$(\eta_1/2) \cdot (N_2/\eta_2)$
E (трудоёмкость)	12 284 элем. реш.	$D \cdot V$
B (оценка ошибок)	0,18	$E^{(2/3)} / 3000$

Полученные значения указывают на умеренную сложность метода: для критически важной операции значение $B = 0,18$ (вероятностная оценка наличия не более одной ошибки) находится в приемлемом диапазоне и подтверждает целесообразность выбранного разделения на слои.

Совокупное применение функционального и модульного тестирования вместе с количественной оценкой кода обеспечивает контроль качества разрабатываемой серверной части на всех уровнях детализации.

Заключение

В ходе ВКР спроектирована и реализована отказоустойчивая микросервисная платформа для интернет-магазина с модулем внутреннего кошелька. В работе обосновано применение паттернов Saga, Transactional Outbox и CloudEvents 1.0, что обеспечило надежное событийное взаимодействие. Спроектированная архитектура базируется на пяти изолированных БД PostgreSQL и микросервисах, разработанных согласно принципам Clean Architecture.

Техническая реализация с использованием Redis и уникальных ограничений СУБД гарантирует идемпотентность операций, а скорость обработки заказа составляет менее одной секунды. Надежность системы подтверждена модульным покрытием (130+ тестов) и метрическим анализом по методу Холстеда ($B = 0,18$), что свидетельствует об оптимальной структуре и высокой сопровождаемости кода.

Практическая значимость работы заключается в создании готового решения для коммерческих предприятий и образовательных целей. Дальнейшее развитие системы предполагает интеграцию внешних платежных шлюзов (ЮKassa, СБП), внедрение OpenTelemetry для мониторинга и переход к оркестрации в среде Kubernetes. Все поставленные цели достигнуты, задачи решены в полном объеме.

Список использованных источников

1. Data Insight. Российский рынок интернет-торговли в 2024 году : аналитический отчёт / Data Insight. – М. : Data Insight, 2025. – 84 с.
2. Фаулер, М. Микросервисы / М. Фаулер. – СПб. : Питер, 2023. – 416 с.
3. Ричардсон, К. Микросервисы. Паттерны разработки и рефакторинга / К. Ричардсон. – М. : Манн, Иванов и Фербер, 2020. – 544 с.
4. Ньюмен, С. Создание микросервисов / С. Ньюмен. – 2-е изд. – СПб. : Питер, 2022. – 624 с.
5. Cloud Native Computing Foundation. CloudEvents 1.0 Specification [Электронный ресурс]. URL: <https://github.com/cloudevents/spec/blob/v1.0.2/cloudevents/spec.md> (дата обращения: 19.05.2025).
6. Буч, Г. Язык UML 2.0. Руководство пользователя / Г. Буч, Дж. Рамбо, А. Якобсон. – М. : ДМК-Пресс, 2021. – 432 с.
7. Apache Software Foundation. Apache Kafka Documentation [Электронный ресурс]. URL: <https://kafka.apache.org/documentation/> (дата обращения: 19.05.2025).
8. Орлов, С. А. Технологии разработки программного обеспечения / С. А. Орлов. – СПб. : Питер, 2020. – 624 с.
9. Дейт, К. Дж. Введение в системы баз данных / К. Дж. Дейт. – 8-е изд. – М. : Вильямс, 2006. – 1328 с.
10. Карвин, Б. Программирование баз данных. SQL-антипаттерны / Б. Карвин. – М. : ДМК-Пресс, 2021. – 336 с.
11. Таненбаум, Э. Распределённые системы. Принципы и парадигмы / Э. Таненбаум, М. Ван Стен. – СПб. : Питер, 2022. – 880 с.
12. Фаулер, М. Корпоративные шаблоны разработки приложений / М. Фаулер. – М. : Вильямс, 2018. – 544 с.

13. Гамма, Э. Приёмы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма [и др.]. – СПб. : Питер, 2020. – 368 с.
14. Кеннеди, М. Concurrency и параллелизм в Python / М. Кеннеди. – М. : ДМК-Пресс, 2023. – 384 с.
15. Halstead, M. H. Elements of Software Science / M. H. Halstead. – New York : Elsevier, 1977. – 128 p.
16. PostgreSQL Global Development Group. PostgreSQL 18 Documentation [Электронный ресурс]. URL: <https://www.postgresql.org/docs/18/> (дата обращения: 19.05.2025).
17. Docker Inc. Docker Documentation [Электронный ресурс]. URL: <https://docs.docker.com/> (дата обращения: 19.05.2025).
18. Redis Ltd. Redis Documentation [Электронный ресурс]. URL: <https://redis.io/docs/> (дата обращения: 19.05.2025).
19. Vercel Inc. Next.js Documentation [Электронный ресурс]. URL: <https://nextjs.org/docs> (дата обращения: 19.05.2025).
20. Redpanda Data Inc. Redpanda Documentation // Redpanda Data Inc. [Электронный ресурс]. URL: <https://docs.redpanda.com/> (дата обращения: 19.05.2025).

Приложение А
(справочное)
Глоссарий

API-Gateway – Микросервис – единая точка входа в систему; выполняет JWT-проверку и проксирование REST-запросов в целевые сервисы.

CloudEvents – Открытая спецификация CNCF, описывающая стандартный конверт события с обязательными атрибутами specversion/id/source/type.

Choreography saga – Разновидность саги без центрального оркестратора; сервисы реагируют на события друг друга через брокер.

CQRS – Command Query Responsibility Segregation – архитектурный шаблон с разделением моделей записи и чтения.

E2E (end-to-end) – Тип тестирования, имитирующий сквозной бизнес-сценарий через все слои системы.

FastAPI – Современный веб-каркас Python для построения API на базе аннотаций типов и Pydantic.

Idempotency-Key – Заголовок HTTP-запроса, по которому сервер распознаёт повторный запрос и не выполняет операцию дважды.

JWT – JSON Web Token – компактный токен авторизации, подписанный криптографически.

Kafka – Распределённый брокер сообщений с фиксированным порядком в разделах (partitions) и группами потребителей.

MinIO – S3-совместимое объектное хранилище, разворачиваемое локально или в облаке.

Outbox pattern – Шаблон, обеспечивающий атомарность записи бизнес-данных в БД и публикации события через буферную таблицу outbox.

Pydantic – Библиотека Python для валидации и сериализации данных на основе аннотаций типов.

Redpanda – Совместимый с Kafka API брокер сообщений, написанный на C++ и не требующий ZooKeeper.

Saga – Последовательность локальных транзакций в нескольких сервисах, связанных событиями, с компенсирующими операциями при сбоях.

SETNX – Команда Redis SET IF NOT EXISTS – атомарная установка ключа со значением, если ключ ещё не существует.

SQLAlchemy – ORM-библиотека Python с поддержкой синхронного и асинхронного режимов.

Storefront BFF – Backend-for-Frontend – серверный слой для конкретного фронтенд-приложения (storefront).

Token theft detection – Механизм обнаружения попыток повторного использования отозванного refresh-токена.

Transactional outbox – См. Outbox pattern.

YooKassa – Российская платёжная система; в данной работе используется мок-реализация.

Приложение Б
(рекомендуемое)
Объекты обращения к базам данных

Pydantic-модели входа/выхода и доменные DTO микросервисов order-service и payment-service.

Листинг Б.1 – DTO OrderCreateIn / OrderOut

```
class OrderItemIn(BaseModel):
    product_id: UUID
    quantity: PositiveInt

class OrderCreateIn(BaseModel):
    items: list[OrderItemIn]
    phone: str = Field(..., pattern=r"^\+7\d{10}$")
    address: str = Field(..., min_length=1, max_length=255)

class OrderOut(BaseModel):
    id: UUID
    status: OrderStatus
    total_amount: Decimal
    created_at: datetime
    items: list[OrderItemOut]
```

Листинг Б.2 – CloudEvent для payment.confirmed

```
class PaymentConfirmedData(BaseModel):
    payment_id: UUID
    order_id: UUID
    amount: Decimal
    confirmed_at: datetime

class PaymentConfirmedEvent(CloudEvent[PaymentConfirmedData]):
    type: Literal["payment.confirmed"] = "payment.confirmed"
    source: Literal["/payment"] = "/payment"
```

Приложение В
(рекомендуемое)
SQL-скрипты создания таблиц

Листинг В.1 – Создание таблицы orders

```

CREATE TYPE order_status AS ENUM (
    'pending', 'paid', 'cancelled', 'refunded', 'delivered'
);

CREATE TABLE orders (
    id                UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id           UUID NOT NULL,
    status            order_status NOT NULL DEFAULT 'pending',
    total_amount      NUMERIC(12, 2) NOT NULL,
    idempotency_key   VARCHAR(64) NOT NULL,
    paid_at           TIMESTAMPTZ,
    created_at        TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    CONSTRAINT uk_orders_idempotency UNIQUE (user_id,
    idempotency_key)
);

CREATE INDEX ix_orders_user ON orders (user_id, created_at DESC);

```

Листинг В.2 – Создание таблицы outbox

```

CREATE TABLE outbox (
    id                BIGSERIAL PRIMARY KEY,
    ce_id             UUID NOT NULL UNIQUE,
    ce_type           VARCHAR(255) NOT NULL,
    ce_source         VARCHAR(255) NOT NULL,
    payload           JSONB NOT NULL,
    headers           JSONB,
    attempts          INT NOT NULL DEFAULT 0,
    last_error        TEXT,
    published_at      TIMESTAMPTZ,
    created_at        TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

CREATE INDEX ix_outbox_unpublished ON outbox (created_at) WHERE
published_at IS NULL;

```

Листинг В.3 – Создание таблицы payments

```

CREATE TYPE payment_status AS ENUM (
    'pending', 'requested', 'succeeded', 'failed', 'refunded'
);

CREATE TABLE payments (
    id                UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    order_id          UUID NOT NULL,
    amount            NUMERIC(12, 2) NOT NULL,
    currency          VARCHAR(3) NOT NULL DEFAULT 'RUB',
    status            payment_status NOT NULL DEFAULT 'pending',
    provider          TEXT, -- wallet | card | sbp | yoomoney

```

```

    idempotency_key    VARCHAR(64) NOT NULL,
    wallet_tx_id       UUID,
    created_at         TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    confirmed_at       TIMESTAMPTZ,
    CONSTRAINT ck_provider CHECK (provider IS NULL OR
        provider IN ('wallet','card','sbp','yoomoney'))
);

```

```

CREATE UNIQUE INDEX uk_payments_idempotency ON payments
(idempotency_key);
CREATE INDEX ix_payments_order ON payments (order_id);

```

Листинг В.4 – Создание таблиц wallets и wallet_transactions

```

CREATE TABLE wallets (
    user_id            UUID PRIMARY KEY,
    balance            NUMERIC(12, 2) NOT NULL DEFAULT 0,
    currency           TEXT NOT NULL DEFAULT 'RUB',
    created_at         TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at         TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    CONSTRAINT ck_wallets_balance_nonneg CHECK (balance >= 0)
);

CREATE TABLE wallet_transactions (
    id                 UUID PRIMARY KEY,
    user_id            UUID NOT NULL,
    kind               TEXT NOT NULL,
    amount             NUMERIC(12, 2) NOT NULL,
    balance_after      NUMERIC(12, 2) NOT NULL,
    idempotency_key    TEXT NOT NULL,
    payment_id         UUID,
    order_id           UUID,
    description        TEXT,
    created_at         TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    CONSTRAINT ck_wallet_tx_kind CHECK (
        kind IN ('debit','credit','refund','signup_bonus','topup')
    )
);

CREATE UNIQUE INDEX uk_wallet_tx_idem ON wallet_transactions
(idempotency_key);
CREATE INDEX ix_wallet_tx_user ON wallet_transactions (user_id,
created_at DESC);

```

Приложение Г
(рекомендуемое)
Структуры слоя представления данных

Листинг Г – Маршрутизация order-service

```

from fastapi import APIRouter, Depends, Header, status
from .deps import get_order_service
from .schemas import OrderCreateIn, OrderOut

router = APIRouter(tags=["orders"])

@router.post("", status_code=status.HTTP_201_CREATED,
response_model=OrderOut)
async def create_order(
    body: OrderCreateIn,
    idempotency_key: str = Header(..., alias="Idempotency-Key"),
    service = Depends(get_order_service),
    user_id: UUID = Depends(get_current_user_id),
) -> OrderOut:
    return await service.create_order(
        user_id=user_id, body=body,
        idempotency_key=idempotency_key,
    )

@router.get("/{order_id}", response_model=OrderOut)
async def get_order(order_id: UUID, service =
Depends(get_order_service)) -> OrderOut:
    return await service.get_by_id(order_id)

```

Приложение Д
(рекомендуемое)
Структуры слоя бизнес-логики

Листинг Д.1 – Идемпотентный debit в WalletServiceImpl

```
class WalletServiceImpl:
    async def debit(
        self, *, user_id: UUID, amount: Decimal,
        idempotency_key: str, payment_id: UUID | None = None,
    ) -> Wallet:
        existing = await self.repo.find_tx_by_idem(idempotency_key)
        if existing:
            return await self.repo.get_wallet(user_id)

        async with self.session.begin():
            wallet = await self.repo.get_for_update(user_id)
            if wallet.balance < amount:
                raise WalletInsufficientError(wallet.balance,
amount)

            wallet.balance -= amount
            tx = WalletTransaction(
                id=uuid4(), user_id=user_id, kind='debit',
                amount=amount, balance_after=wallet.balance,
                idempotency_key=idempotency_key,
payment_id=payment_id,
            )
            self.session.add(tx)
            await self.outbox.append(
                ce_type='wallet.debited',
                payload=WalletDebitedData.from_tx(tx).dict(),
            )
            return wallet
```

Листинг Д.2 – Метод OrderService.create_order

```
class OrderService:
    def __init__(self, session, redis, outbox, stock_client):
        self.session = session
        self.redis = redis
        self.outbox = outbox
        self.stock = stock_client

    async def create_order(self, *, user_id, body,
idempotency_key):
        cache_key = f"order:idem:{user_id}:{idempotency_key}"
        if cached := await self.redis.get(cache_key):
            return await self._reload(UUID(cached))

        await self.stock.validate(items=body.items)

        async with self.session.begin():
            order = Order(
                user_id=user_id,
```

```

        status=OrderStatus.PENDING,
        total_amount=sum(it.unit_price * it.quantity for it
in body.items),
        idempotency_key=idempotency_key,
    )
    self.session.add(order)
    await self.session.flush()
    for it in body.items:
        self.session.add(OrderItem(order_id=order.id,
**it.dict()))
    await self.outbox.append(
        ce_type="order.created",
        payload=OrderCreatedData.from_orm(order).dict(),
    )
    await self.redis.setex(cache_key, 86400, str(order.id))
    return order

```

Приложение Е
(рекомендуемое)
Структуры слоя обращения к базе данных

Листинг Е – ORM-модель Order

```

class Order(Base):
    __tablename__ = "orders"
    id: Mapped[UUID] = mapped_column(primary_key=True,
default=uuid4)
    user_id: Mapped[UUID] = mapped_column(nullable=False)
    status: Mapped[OrderStatus] = mapped_column(
        SQLAlchemyEnum(OrderStatus, name="order_status"), nullable=False,
    )
    total_amount: Mapped[Decimal] = mapped_column(Numeric(12, 2))
    idempotency_key: Mapped[str] = mapped_column(String(64))
    paid_at: Mapped[datetime | None] =
mapped_column(TIMESTAMP(timezone=True))
    created_at: Mapped[datetime] = mapped_column(
        TIMESTAMP(timezone=True), server_default=func.now()
    )
    items: Mapped[list["OrderItem"]] = relationship(
        back_populates="order", cascade="all, delete-orphan",
    )

    __table_args__ = (
        UniqueConstraint("user_id", "idempotency_key",
name="uk_orders_idempotency"),
    )

```

Приложение Ж
(рекомендуемое)
Листинг модульных тестов

Листинг Ж.1 – Тестовая сессия для testcontainers

```
@pytest.fixture(scope="session")
async def pg_container() -> AsyncGenerator[PostgresContainer,
None]:
    with PostgresContainer("postgres:18-alpine") as pg:
        url = pg.get_connection_url().replace("psycopg2",
"asyncpg")
        engine = create_async_engine(url)
        async with engine.begin() as conn:
            await conn.run_sync(Base.metadata.create_all)
        yield pg
        await engine.dispose()
```

Листинг Ж.2 – Интеграционный тест саги

```
@pytest.mark.asyncio
async def test_e2e_saga_order_to_paid(http_client, kafka_consumer):
    resp = await http_client.post(
        "/api/v1/orders",
        json={"items": [{"product_id": str(PRODUCT_ID), "quantity":
2}],
            "phone": "+79991234567", "address": "Test"},
        headers={"Idempotency-Key": "test-key-1"},
    )
    assert resp.status_code == 201
    order_id = resp.json()["id"]
    await wait_for_event(kafka_consumer,
ce_type="payment.confirmed", timeout=2.0)
    order = await http_client.get(f"/api/v1/orders/{order_id}")
    assert order.json()["status"] == "paid"
```

Приложение И
(рекомендуемое)
Спецификация атрибутов таблиц

Таблица И.1 – Спецификация атрибутов таблицы orders

Атрибут	Тип	Ограничения	Назначение
id	UUID	PK, DEFAULT gen_random_uuid()	Идентификатор заказа
user_id	UUID	NOT NULL	Владелец заказа
status	order_status	NOT NULL, DEFAULT 'pending'	Текущее состояние
total_amount	NUMERIC(12,2)	NOT NULL	Полная стоимость заказа
idempotency_key	VARCHAR(64)	NOT NULL	Ключ идемпотентности
paid_at	TIMESTAMPTZ	NULL	Время оплаты
created_at	TIMESTAMPTZ	NOT NULL, DEFAULT NOW()	Время создания заказа

Таблица И.2 – Спецификация атрибутов таблицы payments

Атрибут	Тип	Ограничения	Назначение
id	UUID	PK	Идентификатор платежа
order_id	UUID	NOT NULL	Связанный заказ
amount	NUMERIC(12,2)	NOT NULL	Сумма к оплате
currency	VARCHAR(3)	NOT NULL DEFAULT 'RUB'	Код валюты
status	payment_status	NOT NULL	Состояние платежа
idempotency_key	VARCHAR(64)	UNIQUE NOT NULL	Дедупликация платежей
provider_payment_id	VARCHAR(64)	NULL	Идентификатор у платёжного провайдера
created_at	TIMESTAMPTZ	NOT NULL DEFAULT NOW()	Момент создания
confirmed_at	TIMESTAMPTZ	NULL	Момент финализации

Приложение К
(рекомендуемое)
Иллюстрационные материалы

В данном приложении приведены увеличенные версии ключевых диаграмм для удобства просмотра при защите выпускной квалификационной работы.

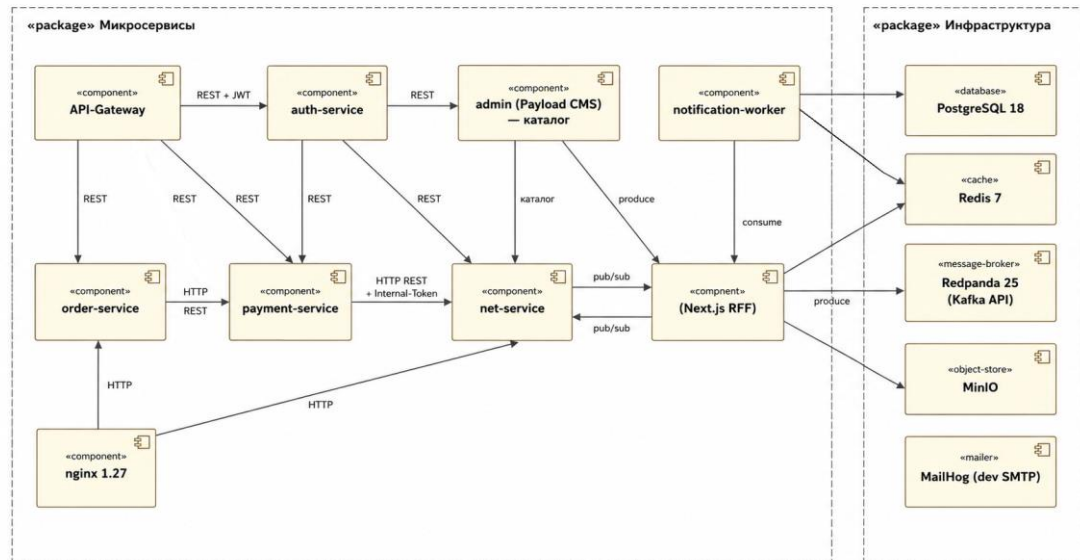


Рисунок К.1 – Диаграмма компонентов

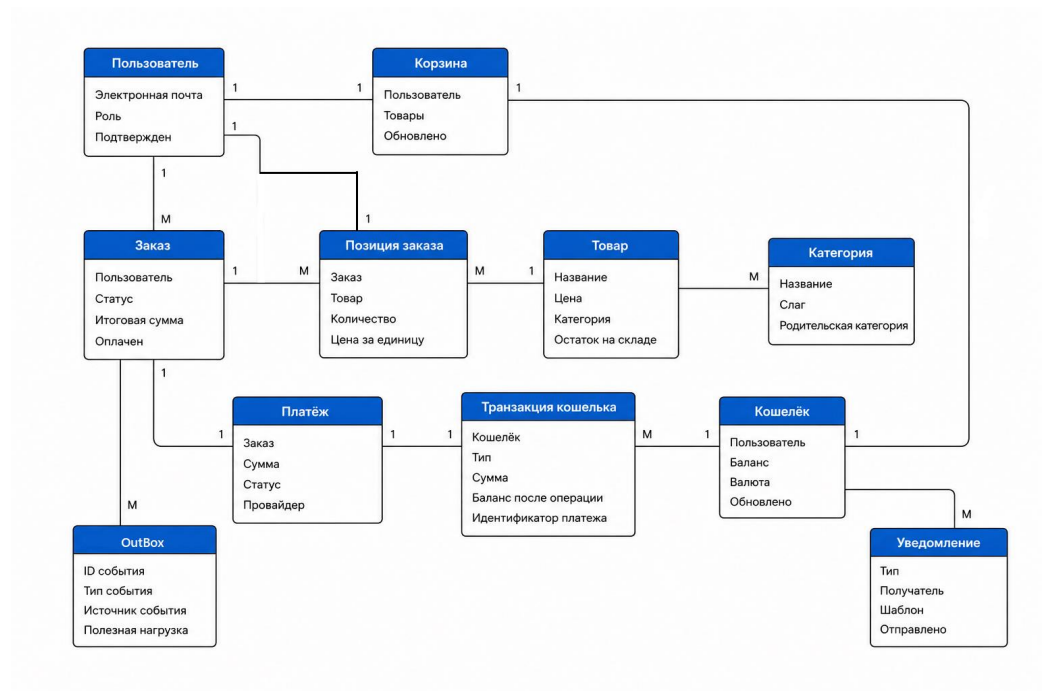


Рисунок К.2 – Концептуальная модель данных

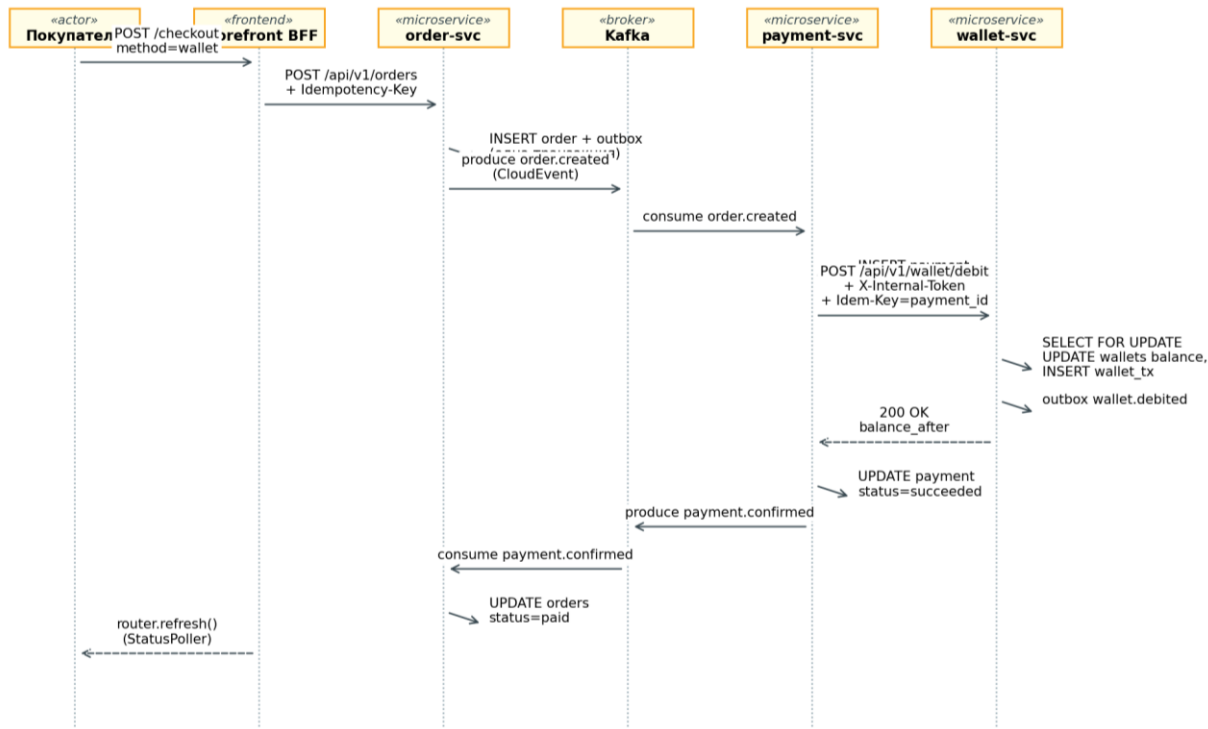


Рисунок К.3 – Хореографическая сага оплаты

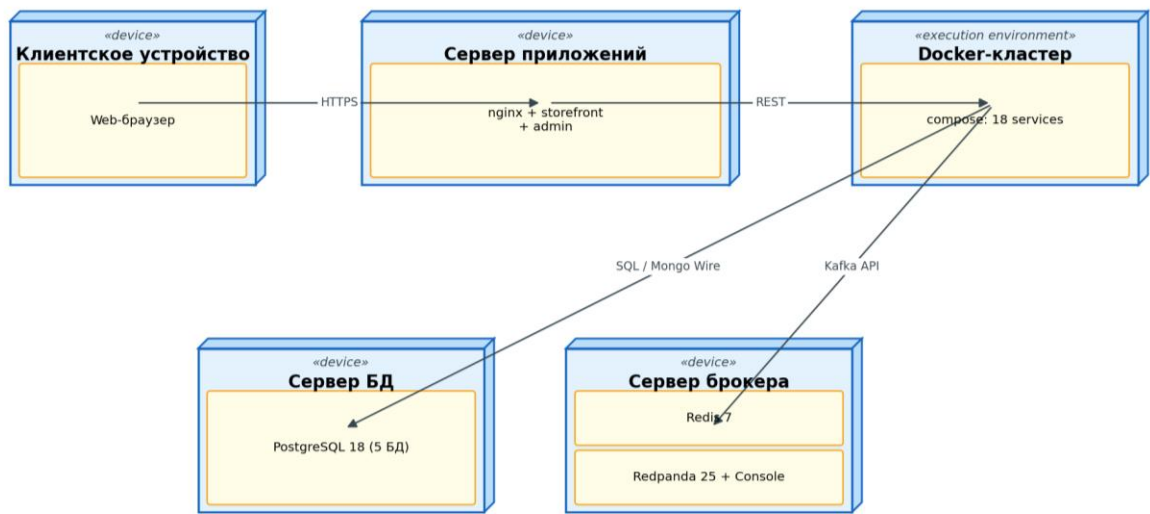


Рисунок К.4 – Диаграмма развёртывания