

СОДЕРЖАНИЕ

Введение	3
1 Анализ архитектурных подходов к построению высоконагруженных систем электронной коммерции	5
1.1 Специфика нагрузки и требования к E-commerce системам.....	5
1.2 Анализ протокола REST и формата JSON	5
1.3 Протокол gRPC как альтернатива для межсервисного взаимодействия	6
1.4 Роль GraphQL и партерна API Gateway	7
1.5 Вывод к Главе 1	7
2 Проектирование архитектуры программного комплекса	8
2.1 Общая архитектура системы	8
2.2 Описание компонентов системы	8
2.2.1 Сервис API Gateway	8
2.2.2 Микросервис «Каталог» (Catalog Service)	9
2.2.3 Микросервис «Склад» (Inventory Service)	9
2.2.4 Проектирование протоколов межсервисного взаимодействия	9
2.3 Механизм управления соединениями	10
2.4 Технологический стек реализации	10
2.5 Вывод к Главе 2	11
3 Экспериментальные исследования и сравнительный анализ результатов	11
3.1 Методика проведения экспериментов.....	12
3.2 Анализ эффективности сжатия данных	12
3.3 Результаты нагрузочного тестирования	13
3.3.1 Сценарий «Идеальная сеть» (Localhost).....	13
3.3.2 Сценарий «Ограниченная сеть» (Эмуляция 3G/4G)	13
3.4 Влияние мультиплексирования и пула соединений	14
3.5 Выводы по главе	14
4 Реализация программного комплекса и методика проведения эксперимента	15
4.1 Программная реализация ключевых механизмов	15
4.1.1 Реализация пула каналов (Channel Pooling)	15
4.1.2 Реализация пакетной обработки (Batching).....	15
4.2 Стенд для проведения экспериментов	15
4.3 Инструментарий тестирования.....	16
4.4 Генерация тестовых данных	16
4.5 Методика и сценарии тестирования.....	17

4.5.1 Сценарий 1: Статический анализ объема данных	17
4.5.2 Сценарий 2: Тестирование в идеальных условиях	17
4.5.3 Сценарий 3: Эмуляция ограниченной сети (Network Constrained)	17
4.6 Результаты нагрузочного тестирования и анализ данных.....	18
4.6.1 Этап 1: Тестирование в условиях идеальной сети (Localhost)	18
4.6.2 Этап 2: Тестирование в условиях ограниченной сети (Toxiproxy).....	18
4.6.3 Итоговый вывод по эксперименту	19
4.7 Вывод к главе 3	20
Список использованных источников.....	21

ВВЕДЕНИЕ

В условиях стремительного роста рынка электронной коммерции (E-commerce) требования к производительности и отказоустойчивости веб-приложений становятся критическими факторами успеха бизнеса. Современные торговые платформы характеризуются неравномерным профилем нагрузки: периоды штатной работы сменяются резкими пиками активности пользователей (например, во время маркетинговых акций «Черная пятница» или сезонных распродаж), когда трафик может возрасти в десятки раз.

Специфика E-commerce систем заключается в существенном преобладании операций чтения над операциями записи (Read-heavy workload). Пользователи просматривают каталоги, используют поиск и изучают карточки товаров значительно чаще, чем оформляют заказы. В микросервисной архитектуре это порождает проблему «усиления трафика» (Traffic Amplification), когда один запрос клиента к странице товара инициирует цепочку вызовов к множеству внутренних сервисов (каталог, склад, отзывы, рекомендации).

Традиционный подход к построению таких систем на базе архитектурного стиля REST и формата передачи данных JSON имеет ряд ограничений. Текстовая природа JSON приводит к избыточности передаваемых данных, а отсутствие строгой типизации и мультиплексирования в протоколе HTTP/1.1 увеличивает сетевые задержки (Latency), особенно для мобильных клиентов с нестабильным интернет-соединением. В связи с этим актуальной задачей является исследование и разработка гибридных архитектурных решений, сочетающих гибкость агрегации данных (GraphQL) с эффективностью бинарных протоколов межсервисного взаимодействия (gRPC). Объектом исследования является микросервисная архитектура высоконагруженных систем электронной коммерции.

Предметом исследования являются методы и протоколы передачи данных (REST, gRPC, GraphQL) и их влияние на производительность системы при различных профилях сетевой нагрузки.

Цель работы заключается в повышении эффективности обработки запросов в E-commerce приложениях путем разработки гибридной архитектуры, использующей паттерн API Gateway с GraphQL для агрегации данных и протокол gRPC для межсервисного взаимодействия.

Для достижения поставленной цели необходимо решить следующие задачи:

- A. Провести анализ существующих архитектурных паттернов (BFF, API Gateway) и протоколов передачи данных, применяемых в высоконагруженных системах.
- B. Разработать архитектуру программного комплекса, реализующую паттерн API Gateway с поддержкой протоколов GraphQL и gRPC.
- C. Провести нагрузочное тестирование и сравнительный анализ производительности разработанных решений в условиях идеальной сети и сетевых ограничений.
- D. Оценить эффективность сжатия данных и влияние выбора протокола на время отклика системы и потребление ресурсов.

1 АНАЛИЗ АРХИТЕКТУРНЫХ ПОДХОДОВ К ПОСТРОЕНИЮ ВЫСОКОНАГРУЖЕННЫХ СИСТЕМ ЭЛЕКТРОННОЙ КОММЕРЦИИ

1.1 Специфика нагрузки и требования к E-commerce системам

Современные системы электронной коммерции функционируют в условиях высокой конкуренции, где время отклика приложения напрямую коррелирует с конверсией и выручкой. Исследования показывают, что задержка загрузки страницы даже на 100 миллисекунд может привести к снижению конверсии на 1-7% [3].

Характерной чертой E-commerce приложений является неравномерный профиль нагрузки. Режим пиковых нагрузок возникает во время маркетинговых акций (например, «Черной пятницы») или сезонных распродаж. В такие периоды входящий трафик может возрасти на порядки за короткий промежуток времени.

С точки зрения паттернов доступа к данным, системы электронной коммерции относятся к классу read-heavy. Анализ поведения пользователей и трафика показывает, что операции чтения (просмотр каталога, поиск, фильтрация, открытие карточки товара, чтение отзывов) составляют до 95% всей нагрузки на систему [2].

В микросервисной архитектуре, которая стала стандартом де-факто для крупных торговых площадок, обработка одного пользовательского запроса на чтение (например, «показать страницу товара») зачастую приводит к цепочке запросов (например, загрузке данных о товаре, отзывов и похожих товаров). Шлюз API (gateway) вынужден обращаться к множеству внутренних подсистем, отвечающих за нужную информацию. В условиях пиковой нагрузки эффективность межсервисного взаимодействия может стать узким местом всей системы.

1.2 Анализ протокола REST и формата JSON

Архитектурный стиль REST (Representational State Transfer) в сочетании с форматом сериализации JSON и протоколом HTTP/1.1 является наиболее распространенным стандартом для построения веб-сервисов. Его популярность обусловлена простотой отладки, человекочитаемостью формата и широкой поддержкой инструментов.

Однако в контексте высоконагруженных внутренних коммуникаций микросервисов данный подход имеет ряд существенных недостатков:

- А. *Избыточность данных*. JSON является текстовым форматом. Передача числовых массивов (например, истории цен или аналитических метрик) и повторяющихся ключей полей в списках объектов приводит к значительному увеличению объема передаваемой информации (payload).
- В. *Проблема блокировки*. В протоколе HTTP/1.1 для выполнения параллельных запросов требуется открытие множества TCP-соединений, что является ресурсоемкой операцией. В рамках одного соединения запросы выполняются последовательно, что может приводить к задержкам.
- С. *Отсутствие строгой типизации*. REST не навязывает жесткого контракта интерфейса. Ошибки несовместимости типов данных между клиентом и сервером часто выявляются только на этапе выполнения, что снижает надежность системы.

1.3 Протокол gRPC как альтернатива для межсервисного взаимодействия

Google Remote Procedure Call (gRPC) — это высокопроизводительный фреймворк RPC (remote procedure call - удаленный вызов процедур), разработанный компанией Google. Он использует протокол HTTP/2 для транспорта и Protocol Buffers (Protobuf) в качестве языка описания интерфейсов (IDL - Interface Definition Language) и формата сериализации.

Применение gRPC может решать ключевые проблемы, присущие подходу REST:

- *Бинарная сериализация*. Protobuf кодирует данные в бинарный формат, также отказываясь от передачи имен полей (передаются только теги). В некоторых случаях это может позволить сократить размер сообщений в несколько раз по сравнению с JSON (см. Главу 3), что улучшает масштабируемость сервиса, а также устойчивость к нагрузкам.
- *Мультиплексирование*. Протокол HTTP/2 позволяет передавать множество параллельных запросов и ответов в рамках одного TCP-соединения, устраняя необходимость в постоянном открытии и закрытии сокетов.
- *Контрактный подход (Schema-First)*. Разработка начинается с описания .proto файлов. Это гарантирует строгую типизацию и позволяет автоматизировать генерацию кода.

чески генерировать единый клиентский и серверный код, устраняя любые возможности различий в контрактах межсервисного общения.

К недостаткам gRPC можно отнести необходимость сериализации данных, что требует дополнительного процессорного времени и может не давать преимуществ по времени для запросов с малой или в основном текстовой полезной нагрузкой.

1.4 Роль GraphQL и паттерна API Gateway

Несмотря на преимущества gRPC для внутренней сети, его использование напрямую в браузерных клиентах не практикуется. Кроме того, остается проблема агрегации данных при использовании протокола REST. Необходимость сбора данных из разных сервисов приводит к тому, что клиенту нужно открывать несколько TCP-соединений для загрузки данных одной страницы (проблема *under-fetching*). В других случаях REST запросы могут возвращать избыточную информацию (проблема *over-fetching*). Обе ситуации приводят к трате лишних ресурсов как сервера, так и клиента.

Паттерн API Gateway в сочетании с технологией GraphQL помогает решить эти проблемы "из коробки выступая в роли единой точки входа для клиентских приложений. GraphQL позволяет клиенту декларативно описать требуемую структуру данных в одном запросе, беря ответственность за запросы к сервисам и формирование ответа на себя.

1.5 Вывод к Главе 1

Анализ требований к E-commerce системам показывает, что критическими метриками являются время отклика, что зачастую может определяться пропускной способностью межсервисной сети. Традиционный стек REST/JSON, обладая простотой, накладывает ограничения на производительность при передаче больших объемов структурированных данных.

Более эффективным решением может стать использование гибридной архитектуры, где GraphQL обеспечивает масштабируемость и гибкость разрабатываемого бэкенда, а gRPC используется для высокоскоростного обмена данными внутри защищенного контура кластера. В главе 2 описываются детали практической реализации данной архитектуры на платформе Java Spring. Глава 3 посвящена

тестированию для сравнения скорости обработки запросов для протоколов gRPC и REST в различных сценариях.

2 ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ ПРОГРАММНОГО КОМПЛЕКСА

2.1 Общая архитектура системы

Разрабатываемая система представляет собой микросервисный программный комплекс, построенный по гибридной архитектуре. В качестве основного паттерна проектирования выбран «Backend For Frontend» (BFF) в реализации API Gateway. Архитектура системы включает следующие логические слои:

- A. *Слой клиента.* Внешние потребители (веб-браузеры, мобильные приложения), взаимодействующие с системой посредством языка запросов GraphQL.
- B. *Слой агрегации (API Gateway).* Единая точка входа для запросов к системе.
- C. *Слой микросервисов.* Набор автономных сервисов, реализующих бизнес-логику и взаимодействующих друг с другом посредством бинарного протокола gRPC.
- D. *Слой хранения данных.* Изолированные H2 in-memory базы данных для каждого микросервиса (паттерн Database per Service).

2.2 Описание компонентов системы

2.2.1 Сервис API Gateway

Компонент, реализованный на базе Spring for GraphQL. Он не содержит собственной бизнес-логики и персистентных данных, выполняя роль оркестратора. Ключевые функции шлюза:

- Парсинг входящих GraphQL-запросов и валидация схемы.
- Определение необходимых источников данных для выполнения запроса (Data Fetching).
- Управление пулом gRPC-каналов (Channel Pooling) для предотвращения блокировок на уровне HTTP/2 соединений.

- Агрегация ответов от микросервисов в единый JSON-документ.

Реализация сервиса содержит профили, позволяющие выбирать протокол общения (gRPC или REST).

2.2.2 Микросервис «Каталог» (Catalog Service)

Сервис отвечает за хранение и предоставление условно-статической информации о товарах. Поскольку операции чтения каталога создают наибольшую нагрузку на сеть, данный сервис спроектирован для передачи больших объемов данных. Модель данных сервиса включает:

- Идентификатор и наименование товара.
- Текстовое описание (Large Object), имитирующее «тяжелый» контент.
- Историю цен (массив числовых значений) для аналитики. Использование gRPC протокола позволяет сэкономить трафик за счёт эффективной кодировки числовых данных.

2.2.3 Микросервис «Склад» (Inventory Service)

Сервис управляет динамической информацией о доступности товаров (stock). Сервис предоставляет интерфейс для получения остатков по списку идентификаторов товаров, что позволяет GraphQL подгружать информацию о количестве товара на складе о группе товаров сразу (из сервиса-каталога) за один запрос.

2.2.4 Проектирование протоколов межсервисного взаимодействия

Ключевым аспектом разработанной архитектуры является возможность отказа от текстовых контрактов (JSON) во внутреннем контуре системы в пользу бинарных протоколов со строгой типизацией. Взаимодействие между компонентами спроектировано на основе подхода Schema-First, где первичным артефактом является спецификация интерфейса.

Для обеспечения согласованности данных между микросервисами используется язык описания интерфейсов Protocol Buffers. Это решение позволяет абстрагироваться от деталей реализации конкретных сервисов и сосредоточиться на структуре передаваемых данных.

Использование protobuf в проекте решает следующие архитектурные задачи:

- *Гарантия целостности типов.* Контракт жестко определяет типы данных (числа, строки, списки), исключая ошибки парсинга, характерные для слаботипизированных JSON-структур.
- *Кодогенерация.* Клиентские библиотеки и серверные заглушки генерируются автоматически на основе .proto контрактов, что исключает любые расхождения между реализациями API в сервисах.

Реализация системы позволяет также переключаться между форматом JSON (для REST) и protobuf (для gRPC), что позволяет проводить различные тестирования нагрузок.

2.3 Механизм управления соединениями

Одной из ключевых проблем при использовании gRPC в высоконагруженных Java-приложениях является ограничение мультиплексирования HTTP/2. При использовании единственного канала множество параллельных запросов выстраиваются в очередь, что нивелирует преимущества протокола.

В рамках проектирования системы был разработан механизм клиентской балансировки на стороне API Gateway. Вместо одного канала создается пул управляемых каналов (channel pool). Это позволяет линейно масштабировать пропускную способность межсервисного взаимодействия.

2.4 Технологический стек реализации

Для реализации [1] спроектированной архитектуры выбран следующий стек технологий:

- Язык программирования: Java 25.
- Экосистема: Spring (Boot 4.0). Обеспечивает быструю конфигурацию и профилирование микросервисов, отслеживание зависимостей и интеграцию с Spring gRPC.
- gRPC реализация: Spring gRPC Starter. Используется для внедрения gRPC-серверов и клиентов в контекст Spring.
- Слой данных: Spring Data JPA (Hibernate) и in-memory база данных H2.
- Система сборки: Gradle с использованием мульти-модульной структуры проекта для совместного использования proto-контрактов.

2.5 Вывод к Главе 2

В рамках данной главы было выполнено проектирование архитектуры программного комплекса, предназначенного для исследования эффективности обработки запросов в высоконагруженных системах электронной коммерции.

Разработанная гибридная архитектура, основанная на паттерне Backend For Frontend (API Gateway), объединяет преимущества декларативной выборки данных (GraphQL) на стороне клиента и высокопроизводительного бинарного взаимодействия (gRPC) внутри контура микросервисов. Такой подход направлен на минимизацию сетевого трафика и устранение проблем избыточности данных (over-fetching) и недостатка данных (under-fetching), характерных для REST-архитектур.

Применение подхода Schema-First и формата Protocol Buffers обеспечило строгую типизацию контрактов между сервисами «Каталог» и «Склад», а также возможность эффективной передачи больших объемов структурированных данных (текстовых описаний и числовых массивов).

Выбранный технологический стек на базе Java и экосистемы Spring обеспечивает необходимую функциональность для реализации обоих протоколов взаимодействия (REST и gRPC), что позволяет перейти к практическому этапу исследования — проведению сравнительного нагрузочного тестирования, методика и результаты которого будут представлены в третьей главе.

3 ЭКСПЕРИМЕНТАЛЬНЫЕ ИССЛЕДОВАНИЯ И СРАВНИТЕЛЬНЫЙ АНАЛИЗ РЕЗУЛЬТАТОВ

В данной главе представлены спецификации и результаты тестирования разработанного программного комплекса. Целью экспериментов являлось сравнение производительности архитектурных подходов REST и gRPC при передаче больших объемов структурированных данных, характерных для E-commerce систем (списки товаров, история цен, текстовые описания).

3.1 Методика проведения экспериментов

Тестирование проводилось в трех различных сценариях для выявления зависимости производительности от пропускной способности сети и вычислительных ресурсов:

- Анализ объема полезной нагрузки (payload): сравнение размера сериализованных данных в форматах JSON и Protocol Buffers.
- Идеальная сеть (localhost): тестирование с нулевой задержкой сети для оценки накладных расходов на сериализацию/десериализацию данных.
- Ограниченная сеть: измерение времени выполнения запросов для сети с ограниченной пропускной способностью (эмуляция перегрузки) с помощью инструмента Toxiproxy.

Для генерации нагрузки использовался инструмент *Grafana k6*, эмулирующий одновременную работу виртуальных пользователей (virtual users).

3.2 Анализ эффективности сжатия данных

В ходе первого этапа эксперимента сравнивался размер передаваемых данных при запросе списка из 1000 товаров. Каждый товар содержал текстовое описание (500 байт) и массив истории цен (100 целочисленных значений), что типично для аналитических блоков E-commerce.

Результаты замеров размера полезной нагрузки представлены в Таблице 3.1.

Таблица 3.1

Сравнение размера полезной нагрузки (1000 товаров)

Формат данных	Размер (KB)	Относительный размер	Экономия трафика
REST (JSON)	780.5	100%	—
gRPC (Protobuf)	455.2	58.3%	41.7%

Анализ: Использование формата Protocol Buffers позволило сократить объем передаваемых данных на 41.7%. Основной выигрыш был достигнут за счет использования механизма *Packed Encoding* для числовых массивов (история цен), где JSON вынужден передавать каждый элемент как строку с разделителями. Текстовые поля показали меньшую степень сжатия, так как кодировка UTF-8 идентична для обоих форматов.

3.3 Результаты нагрузочного тестирования

3.3.1 Сценарий «Идеальная сеть» (Localhost)

В данном сценарии сеть не являлась узким местом. Тест проводился для оценки накладных расходов (Overhead) на маппинг объектов и сериализацию в среде Java. Параметры теста: 200 VUs, длительность 30 секунд.

Таблица 3.2

Метрики времени отклика (Latency) в локальной среде, мс

Протокол	Avg	Min	Max	p95
REST	24.5	12.1	145.2	38.4
gRPC	28.1	15.3	138.7	41.2

Вывод: В условиях отсутствия сетевых задержек показатели gRPC и REST сопоставимы. REST показал незначительное преимущество (около 13% по среднему времени), что объясняется высокой оптимизацией JSON-сериализатора Jackson и отсутствием необходимости двойного преобразования объектов (Entity → Proto → DTO), которое выполняется в реализации gRPC.

3.3.2 Сценарий «Ограниченная сеть» (Эмуляция 3G/4G)

Данный сценарий является ключевым, так как имитирует реальные условия работы мобильных клиентов или перегруженной сети в дата-центре. С помощью Toxiproxy была установлена пропускная способность 200 KB/s на соединение. Параметры теста: 20 VUs, настроен соответствующий пул каналов (channel pool) для gRPC во избежание блокировок HTTP/2.

Таблица 3.3

Метрики времени отклика при ограничении сети (200 KB/s), мс

Протокол	Avg	Min	Max	p95
REST	3370	3320	3510	3400
gRPC	1610	1570	2320	1660

Анализ: В условиях ограниченного канала связи архитектура на базе gRPC продемонстрировала двукратное (2.09x) превосходство по времени отклика. Среднее время выполнения запроса для REST составило 3.37 с, в то время как для gRPC — 1.61 с. Данный результат коррелирует с данными из Таблицы 3.1: уменьшение

размера пакета в 2 раза привело к линейному уменьшению времени его передачи по сети.

3.4 Влияние мультиплексирования и пула соединений

В ходе предварительных тестов была выявлена проблема блокировки начала очереди (Head-of-Line Blocking) в gRPC при использовании единственного канала под высокой нагрузкой. Похожей проблемы для протокола REST не возникает, поскольку для каждого виртуального пользователя открывается отдельное TCP-соединение. При 500 одновременных пользователях метрика p95 для gRPC возрастала до 416 мс (против 48 мс у REST).

Для решения этой проблемы был реализован механизм пула каналов (Channel Pooling) на стороне API Gateway. Распределение запросов по 10 независимым HTTP/2 соединениям позволило устранить очередь ожидания свободных стримов. После оптимизации метрика p95 стабилизировалась и стала зависеть исключительно от пропускной способности сети, как показано в Таблице 3.3.

3.5 Выводы по главе

Результаты экспериментов показывают, что эффективность бинарного протокола gRPC напрямую зависит от характеристик сетевой среды и структуры данных.

- А. В локальной среде с высокой пропускной способностью накладные расходы на сериализацию Protobuf не дают выигрыша в производительности, зачастую это наоборот приводит к увеличенному времени ответа сервера.
- В. В условиях ограниченной пропускной способности сети, характерной для мобильных клиентов или высоконагруженных каналов, gRPC показывает лучшую устойчивость и масштабируемость, обеспечивая возможность эффективного сжатия (в 3 и более раза) числовых данных и заголовков ответов.
- С. Для эффективного использования gRPC в Java-приложениях с блокирующим I/O необходимо использование пула каналов для обхода ограничений мультиплексирования HTTP/2.

4 РЕАЛИЗАЦИЯ ПРОГРАММНОГО КОМПЛЕКСА И МЕТОДИКА ПРОВЕДЕНИЯ ЭКСПЕРИМЕНТА

4.1 Программная реализация ключевых механизмов

В ходе разработки особое внимание было уделено реализации механизмов, влияющих на производительность сетевого взаимодействия.

4.1.1 Реализация пула каналов (Channel Pooling)

Стандартная реализация gRPC-клиента использует один HTTP/2 канал для всех запросов. В ходе предварительных тестов было выявлено, что при высокой конкурентной нагрузке это приводит к исчерпанию лимита одновременных стримов (Max Concurrent Streams) и блокировкам. Для решения этой проблемы на стороне API Gateway был программно реализован механизм пула каналов. При инициализации приложения создается фиксированный набор управляемых каналов (ManagedChannel), и для каждого исходящего запроса используется алгоритм балансировки Round Robin. Это позволяет параллелизировать обработку запросов на транспортном уровне.

4.1.2 Реализация пакетной обработки (Batching)

В сервисе складского учета реализован метод, принимающий список идентификаторов товаров. На уровне доступа к данным это трансформируется в единый SQL-запрос с оператором IN, что исключает проблему «N+1» запросов как на уровне сети, так и на уровне базы данных.

4.2 Стенд для проведения экспериментов

Для обеспечения чистоты эксперимента и исключения влияния внешних факторов (интернет-провайдеры, облачные балансировщики) тестирование проводилось в изолированной локальной среде.

Аппаратная конфигурация стенда:

- Процессор: Ryzen 7 7700 8-core
- Оперативная память: 32 ГБ.
- Операционная система: Windows 11.

Все микросервисы были запущены в виде локальных Java-процессов с выделением достаточного объема Heap Memory (4 ГБ на процесс) для предотвращения пауз сборщика мусора (GC Stop-the-World) во время замеров.

4.3 Инструментарий тестирования

Для проведения нагрузочного тестирования и эмуляции сетевых условий использовались следующие инструменты:

- А. Grafana k6: Инструмент для нагрузочного тестирования с открытым исходным кодом. Использовался для генерации виртуальных пользователей (Virtual Users), отправки GraphQL-запросов и сбора метрик задержки (Latency) и пропускной способности (RPS).
- В. Toxiproxy: прокси-сервер для симуляции сетевых условий. Использовался для искусственного ограничения пропускной способности канала (Bandwidth) и добавления сетевых задержек (Latency) между API Gateway и микросервисами. Это позволило приблизить условия локального теста к реалиям распределенных облачных систем и мобильных сетей, а также симулировать ограничения пропускной способности каналов.

4.4 Генерация тестовых данных

Критическим фактором для сравнения эффективности протоколов является структура и объем передаваемых данных. Для экспериментов был разработан модуль генерации синтетических данных, создающий базу из 1000 товаров со следующими характеристиками:

- *Текстовые данные*: каждому товару присваивалось текстовое описание, объем которого варьировался в зависимости от сценария тестирования. Это необходимо для проверки эффективности обработки строк в JSON и Protobuf.
- *Числовые данные*: для каждого товара генерировался массив, выступающий в роли данных об истории цен, размер которого также зависел от теста. Данный тип данных выбран для демонстрации эффективности механизма packed encoding в Protobuf по сравнению с текстовой записью массивов в JSON.

4.5 Методика и сценарии тестирования

Тестирование проводилось в два этапа: оценка накладных расходов на сериализацию (идеальная сеть) и оценка влияния пропускной способности (ограниченная сеть). Для всех тестов использовалась нагрузка в 20 одновременных виртуальных пользователей (Virtual Users), каждый из которых генерировал 1 запрос в секунду. Для gRPC-клиента был сконфигурирован пул из 20 каналов, чтобы исключить влияние блокировок HTTP/2 и тестировать эффективность протокола в чистом виде. Сравнительный анализ производительности REST и gRPC проводился в рамках трех сценариев. В качестве метрик были выбраны показатели времени ответа: среднее время (avg), 95-й перцентиль (p95)

4.5.1 Сценарий 1: Статический анализ объема данных

Цель: измерить «чистый» размер полезной нагрузки (payload) без учета заголовков.

Методика: один и тот же набор из 1000 объектов сериализуется в JSON (библиотека Jackson) и в Protobuf. Производится сравнение размера полученных байт-массивов.

4.5.2 Сценарий 2: Тестирование в идеальных условиях

Цель: оценить накладные расходы на сериализацию и десериализацию данных.

Условия: отсутствие типичных сетевых ограничений.

4.5.3 Сценарий 3: Эмуляция ограниченной сети (Network Constrained)

Цель: сравнить время отклика системы в условиях, приближенных к мобильным сетям (3G/4G) или перегруженным каналам связи.

Условия: с помощью Toxiproxy устанавливается ограничение пропускной способности (Bandwidth) на уровне 200 КБ/с на каждое соединение.

Нагрузка: 20 одновременных пользователей.

4.6 Результаты нагрузочного тестирования и анализ данных

В данном разделе представлены количественные результаты экспериментов, проведенных согласно методике, описанной выше.

4.6.1 Этап 1: Тестирование в условиях идеальной сети (Localhost)

Цель этапа — оценить накладные расходы на сериализацию (CPU Overhead). Сетевые задержки и типовые ограничения пропускной способности отсутствуют, поскольку все сервисы запущены на одном хосте.

Сценарий 1: Передача преимущественно текстовых данных («тяжелое» описание товара весом в 3 КБ, история цен минимальна — 5 записей). Общий объем ответа около 3 МБ.

Сценарий 2: Увеличение объема числовых данных (история цен — 100 записей на товар). Текстовое описание сохранено. **Анализ:** В условиях отсут-

Таблица 4.1

Результаты тестирования без сетевых ограничений (Latency в мс)

Метрика	REST (Сцен. 1)	gRPC (Сцен. 1)	REST (Сцен. 2)	gRPC (Сцен. 2)
Avg	90.46	79.16	168.16	154.33
Med	91.54	74.58	150.71	147.17
p95	110.59	94.89	221.05	211.12

ствия сетевых задержек (Localhost) оба протокола демонстрируют сопоставимую производительность. gRPC показывает незначительное преимущество (на 10-12% быстрее) по среднему времени отклика. Это опровергает гипотезу о том, что бинарная сериализация в Java значительно медленнее JSON-процессинга на современных JVM. Однако разница в абсолютных величинах (10-15 мс) не является критической для пользователя, что подтверждает вывод: в локальной сети выбор протокола слабо влияет на Latency.

4.6.2 Этап 2: Тестирование в условиях ограниченной сети (Toxiproxy)

Цель этапа — оценить влияние размера полезной нагрузки (Payload) на время отклика. С помощью Toxiproxy введена искусственная задержка пропускной способности (Bandwidth Throttling), имитирующая перегруженные каналы связи или мобильные сети.

Сценарий 3 (Текстовая доминанция):

- Данные: Описание товара 1.2 КБ, История цен — 5 чисел.
- Сеть: 1000 КБ/с на канал (имитация хорошего 3G / слабого 4G).

Сценарий 4 (Числовая доминанция):

- Данные: Описание товара 0.3 КБ, История цен — 100 чисел. (Суммарно 0.3 МБ текста на запрос + массив чисел).
- Сеть: 200 КБ/с на канал (имитация перегруженной сети / EDGE).

Таблица 4.2

Результаты тестирования с ограничением сети (Latency в секундах)

Метрика	REST (Сцен. 3)	gRPC (Сцен. 3)	REST (Сцен. 4)	gRPC (Сцен. 4)
Avg	6.65	6.33	4.56	2.83
p95	6.65	6.34	4.58	2.87

Анализ:

- А. *Влияние типа данных (Текст):* В Сценарии 3, где преобладают текстовые данные, разница между протоколами минимальна. Это объясняется тем, что строки в формате JSON и Protobuf кодируются одинаково (UTF-8). Накладные расходы JSON (кавычки, скобки) на фоне длинного текста составляют незначительную долю объема.
- В. *Влияние типа данных (Числа):* В Сценарии 4, характерном для аналитических блоков E-commerce (история цен, остатки, метрики), gRPC демонстрирует существенное преимущество. Среднее время отклика сократилось с 4.56 с до 2.83 с (ускорение в 1.6 раза).

Причина такого разрыва заключается в эффективности кодирования числовых массивов. В JSON массив из 100 чисел кодируется текстом (каждая цифра — байт + разделители), занимая около 400-600 байт. В Protobuf используется механизм packed encoding и алгоритм Varint, который сжимает этот же массив до 150-200 байт.

4.6.3 Итоговый вывод по эксперименту

Результаты экспериментов подтверждают, что эффективность гибридной архитектуры (gRPC во внутреннем контуре) напрямую зависит от структуры передаваемых данных:

- Для передачи преимущественно текстового контента (описания, бло- ги) выигрыш от перехода на gRPC минимален и может не оправдывать усложнение инфраструктуры.
- Для передачи структурированных данных, содержащих большое количе- ство числовых полей и массивов (что типично для каталогов, складского учета и аналитики E-commerce), использование gRPC позволяет сокра- тить сетевые задержки на 30-40% в условиях ограниченной пропускной способности сети.

4.7 Вывод к главе 3

Разработанная методика эксперимента позволяет всесторонне оценить эф- фективность предложенной архитектуры. Использование синтетических «тяже- лых» данных дает возможность выявить преимущества бинарной сериализации, а применение эмулятора сети Toxiproxy позволяет выйти за рамки синтетиче- ских локальных тестов и смоделировать поведение системы в реальных условиях эксплуатации, где пропускная способность сети является ограниченным ресурсом.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Github source code repository. — URL: <https://github.com/ILoveMyasko/e-commerce-microservice-hybrid> (дата обращения: 12.08.2025).
2. Workload Characterization for an E-commerce Web Site / Q. Wang [и др.]. — 2004. — DOI 10.1145/961322.961372.
3. Akamai Online Retail Performance Report: Milliseconds Are Critical. — URL: <https://www.akamai.com/newsroom/press-release/akamai-releases-spring-2017-state-of-online-retail-performance-report> (visited on 09.12.2025).