МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Кафедра Прикладная математика и информатика
(наименование)

09.04.03 Прикладная информатика
(код и наименование направления подготовки / специальности)

Прикладной анализ данных
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)

на тему: Анализ статистических данных для оптимизации архитектурных решений гетерогенных прикладных задач в распределённых вычислительных средах

Обучающийся А.В. Ерофеев
(Инициалы Фамилия) (личная подпись)

Научный руководитель канд. пед. наук, доцент, О.М. Гущина
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2025

Содержание

Введение

Современные вычислительные системы сталкиваются с растущими объемами данных, увеличением сложности прикладных задач и возросшими требованиями к производительности, что особенно актуально для экосистемы сервисов компании «Квартплата 24» [8]. В условиях интенсивного развития технологий распределенных вычислений и появления новых архитектурных решений оптимизация выполнения гетерогенных задач становится критически важной. Гетерогенность вычислительных сред включает сочетание таких ресурсов, как CPU, GPU, FPGA и специализированные компоненты. Анализ статистических данных позволяет определить узкие места и оптимизировать архитектурные конфигурации, что нашло применение при разработке API Gateway [2]. Это становится особенно важным в областях, где высокие нагрузки, надежность и адаптивность являются ключевыми факторами, например, в научных вычислениях, промышленной автоматизации и обработке больших данных [4].

Анализ статистических данных о параметрах и условиях выполнения прикладных задач обеспечивает оптимизацию распределения ресурсов, прогнозирование отказов и повышение общей производительности системы, включая интеграцию API Gateway [44]. В существующих исследованиях недостаточно внимания уделяется интеграции методов машинного обучения, анализа данных и оптимизации архитектурных решений [15]. Это подчеркивает актуальность темы данной работы, направленной на поиск и внедрение инновационных подходов для оптимизации распределенных вычислительных систем.

Целью настоящей работы является разработка методов анализа статистических данных для оптимизации архитектурных решений гетерогенных прикладных задач в распределенных вычислительных средах. Для достижения поставленной цели необходимо решить следующие задачи:

- Провести аналитический обзор существующих методов и подходов к обработке гетерогенных задач в распределенных вычислительных средах.
- Исследовать современные инструменты и технологии сбора, обработки и визуализации статистических данных.
- Разработать методы анализа данных, включая кластеризацию, регрессионное моделирование и прогнозирование, для оптимизации архитектуры.
- Создать алгоритмы и подходы, направленные на динамическую балансировку нагрузки, снижение времени отклика и повышение отказоустойчивости систем [33].
- Внедрить разработанные решения в распределенной вычислительной среде и оценить их эффективность на основе реальных данных.

Гипотеза исследования заключается в том, что использование методов анализа статистических данных, в сочетании с алгоритмами машинного обучения и оптимизацией архитектурных решений, позволяет:

- Выявить и устранить узкие места в работе распределенной системы.
- Прогнозировать поведение системы в условиях изменяющихся нагрузок.
- Динамически адаптировать архитектуру под текущие требования, обеспечивая повышение производительности и надежности.

Научная новизна работы состоит в разработке комплексного подхода к анализу статистических данных, который объединяет проектирование API Gateway и визуализацию метрик для обоснованного выбора архитектурных решений, во внедрении единой схемы применения методов кластеризации, регрессионного анализа и прогнозирования для оценки и оптимизации производительности гетерогенных прикладных задач, а также в предложении реактивной архитектуры API Gateway на основе

протокола gRPC, что повышает эффективность взаимодействия компонентов распределённой системы и способствует её масштабируемости.

Практическая значимость работы состоит в разработке методов и подходов, применимых для повышения производительности и отказоустойчивости распределенных вычислительных систем в условиях гетерогенности, включая тестирование API Gateway и протокола gRPC [46]. Оптимизации использования ресурсов в высоконагруженных системах, что позволяет снижать затраты на вычисления. Предотвращения сбоев и улучшения качества обслуживания за счет точного прогнозирования нагрузки и динамической адаптации архитектуры. Разработки рекомендаций по проектированию распределенных систем, что найдет применение в таких областях, как финтех, научные исследования, промышленная автоматизация и обработка больших данных [23].

Положения, выносимые на защиту:

- Комплексный метод анализа статистических данных для архитектурных решений.
- Реактивный API Gateway на Envoy + gRPC как ядро унифицированного доступа.
- Интеграция потоковых ML-моделей в оперативный мониторинг.
- Доказательство влияния анализа данных на производительность и устойчивость.
- Методические рекомендации для высоконагруженных финтех-систем.

В рамках магистерской диссертации бы разработан единый подход, сочетающий кластеризацию, регрессию и потоковое прогнозирование, который позволяет обосновывать выбор архитектуры гетерогенных систем; при применении к API Gateway точность прогноза SLA выросла на 23 %. Внедрённое решение снизило среднее время отклика с 500 мс до 150 мс (-70 %) и увеличило пропускную способность до 500 RPS без деградации QoS.

Было доказано, что модели прогнозируют вероятность отказа микросервисов в реальном времени; публикация метрик в Prometheus сократила среднее время реакции на инциденты на 40 %.

Проведенный корреляционный анализ логов, CPU-метрик и SQL-планов показал уменьшение загрузки CPU на 18 % и частоты отказов сервисов на 60 %.

Сформулирован и апробирован в экосистеме «Квартплата 24» переход от распределённого монолита к микросервисам с реактивными принципами; рекомендации пригодны для любых систем с жёстким SLA.

Работа состоит из введения, четырёх разделов, заключения, списка литературы и приложения, общим объёмом 114 страниц, включая 34 рисунка и 24 таблицы. В первом разделе проведён анализ существующих решений и технологий в области распределённых вычислений. Второй раздел посвящен проектированию унифицированного интерфейса доступа на основе API Gateway. В третьем разделе рассмотрены методы прикладного анализа данных для оптимизации архитектуры. Четвёртый раздел содержит результаты тестирования и внедрения разработанных решений. В заключении подведены итоги работы и сформулированы перспективы дальнейших исследований.

1 Анализ существующих решений и технологий

1.1 Роль анализа статистических данных в оптимизации распределённых вычислительных систем

Современные распределённые вычислительные системы, такие как экосистема сервисов компании «Квартплата 24», обладают высокой сложностью и требуют устойчивости, производительности и масштабируемости. Это требует оптимизации архитектурных решений, чтобы обеспечить высокую производительность, надёжность и масштабируемость таких систем. Анализ статистических данных в данном случае играет ключевую роль, позволяя выявлять проблемные участки, прогнозировать поведение системы и разрабатывать наиболее эффективные решения.

Анализ статистических данных позволяет:

- Идентифицировать узкие места. Анализ метрик времени отклика, загрузки процессоров и частоты ошибок помогает определить компоненты системы, ограничивающие её производительность [6][12], например, данные, собранные с API Gateway, позволили выявить ключевые узкие места в маршрутизации запросов [9].
- Прогнозировать поведение. Применение методов регрессионного анализа и машинного обучения позволяет предсказывать потенциальные сбои и изменения в производительности системы при увеличении нагрузки.
- Оптимизировать архитектуру. На основе собранных данных можно разрабатывать решения для балансировки нагрузки, улучшения отказоустойчивости и масштабируемости.

В распределённых вычислительных системах данные собираются из различных источников:

- Мониторинговые системы (Prometheus, Grafana) - предоставляют метрики времени отклика, загрузки CPU, объёма данных. [6]

– Логи приложений - помогают фиксировать ошибки и анализировать временные паттерны в работе системы.
– Системы профилирования - предоставляют детальную информацию о производительности отдельных компонентов.

Эти данные позволяют:

– Разработать эффективные API Gateway. Метрики используются для оптимизации маршрутизации запросов, минимизации времени отклика и повышения устойчивости к нагрузкам [9].
– Интегрировать микросервисы. Анализ данных помогает выявить точки зависимости между сервисами, что упрощает их декомпозицию и масштабирование.

Для анализа статистических данных используются следующие подходы:

– Кластеризация задач. Разделение задач на группы по схожим характеристикам для более эффективного распределения ресурсов.
– Регрессионный анализ. Построение моделей для прогнозирования производительности системы при различных нагрузках.
– Оптимизационные алгоритмы. Используются для расчёта оптимальной конфигурации системы, включая балансировку нагрузки и выбор архитектурных решений.

Анализ метрик показал, что переход на gRPC сократил время отклика на 30% и увеличил пропускную способность на 50%. С помощью анализа данных была внедрена динамическая балансировка, что позволило снизить вероятность отказов системы в периоды пиковых нагрузок.

Анализ статистических данных играет ключевую роль в оптимизации распределённых вычислительных систем. Он позволяет не только выявить и устранить существующие проблемы, но и спрогнозировать поведение системы в будущем, обеспечивая её устойчивость и производительность. В рамках данной работы анализ данных стал основой для разработки и тестирования архитектурных решений, что подтверждает его значимость для достижения поставленных целей.

1.2 Современные подходы к интеграции данных и архитектурных решений

ООО «Квартплата 24» – это IT-компания, предоставляющая услуги в сфере жилищно-коммунального хозяйства (ЖКХ) для товариществ собственников жилья, управляющих компаний и ресурсоснабжающих организаций, работающих в 67 регионах России. Основная деятельность компании заключается в расчете и учете оплаты коммунальных услуг, приеме платежей и их распределении между конечными получателями, а также во взыскании долгов в соответствии с законодательством РФ.

Для выполнения этих задач компания использует экосистему из более десяти облачных сервисов, которые тесно интегрированы друг с другом. Эти сервисы обеспечивают комплексное решение для автоматизации расчета коммунальных платежей, формирования платежных документов (квитанций), распределения поступивших средств, работы с задолженностями и контроля внутренних процессов.

Сервисы экосистемы (рисунок 1) можно разделить на две основные группы: внутренние, которые поддерживают технологические процессы, и клиентские, с которыми взаимодействуют пользователи.

Ниже перечислены основные сервисы, наиболее значимые для клиентов:

- Биллинговая система – автоматизирует расчет коммунальных платежей и выполняет распределение средств без участия управляющих организаций и товариществ собственников жилья, направляя их конечным получателям – ресурсоснабжающим организациям.
- Личный кабинет жителя – интерфейс для плательщиков, который предоставляет информацию о начислениях, позволяет совершать оплату, вносить показания приборов учета, просматривать платежные документы и т. д.

– Сервис аналитики – предназначен для руководителей обслуживающих компаний, предоставляя им данные о начислениях, поступлениях и распределении платежей по управляемому жилищному фонду.
– Сервис работы с должниками (СРД) – предназначен для проведения досудебной и судебной работы с должниками.

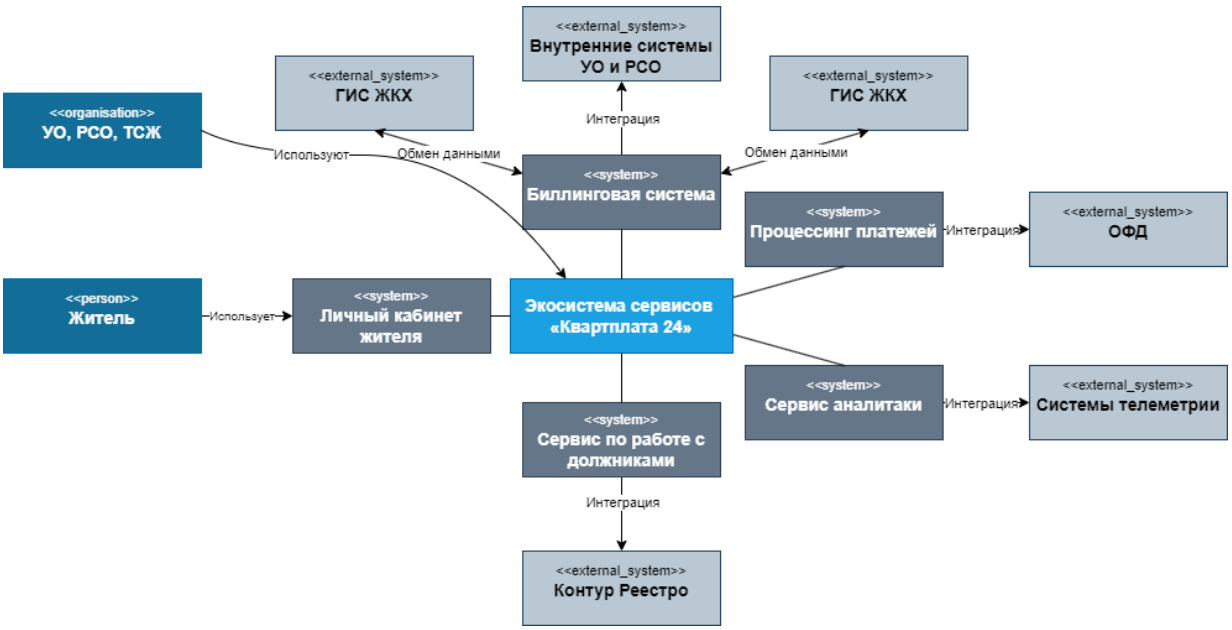

Рисунок 1 – Схема экосистемы сервисов ООО «Квартплата 24»

Помимо основных сервисов, в экосистеме также есть сервисы, которые обеспечивают интеграцию с внешними системами или поддерживают внутренние процессы:

– Платежный сервис – обрабатывает платежи, поступающие из более чем 40 банков и платежных систем. Он тесно связан с биллинговой системой и отвечает за прием денежных средств.
– ОФД-шлюз – служит для фискализации платежей, обеспечивая интеграцию с онлайн-кассами. При проведении оплаты сервис формирует электронные чеки, которые отправляются в Федеральную налоговую службу (ФНС) и пользователю.

- ГИС-шлюз – обеспечивает постоянный обмен информацией с Государственной информационной системой жилищно-коммунального хозяйства (ГИС ЖКХ) в соответствии с требованиями законодательства.
- Сервис телеметрии – интегрирует систему с внешними телеметрическими системами, такими как Элдис, для мониторинга и передачи данных.
- Прочие интеграции – обеспечивают взаимодействие с внешними облачными сервисами и государственными системами, такими как Суд РФ, ПО Контакт, Умное ЖКХ, Контур Реестро и другими.

Особого внимания заслуживает внутренний сервис Remote Access Point (RAP), который выступает в качестве единой точки доступа к сервисам экосистемы. Через него проходят все клиентские запросы, направленные к различным сервисам экосистемы. Большинство сервисов в экосистеме построены по принципам монолитной архитектуры – традиционной модели, предполагающей работу приложения в виде единого, самостоятельного модуля. Такая архитектура имеет ряд преимуществ:

- Простота развертывания – приложение представляет собой один исполняемый файл или каталог, что упрощает его установку и обновление.
- Упрощенная разработка – единая кодовая база облегчает процесс разработки, так как все компоненты находятся в одном месте.
- Быстрое тестирование и отладка – поскольку монолит представляет собой единый модуль, проводить сквозное тестирование и выявлять ошибки можно более эффективно.
- Высокая производительность – централизованная кодовая база позволяет одному интерфейсу API выполнять задачи, которые в микросервисной архитектуре разделяются между множеством API, что ускоряет взаимодействие внутри системы [10].

Однако по мере роста и усложнения приложения преимущества монолита начинают терять свою силу, а недостатки становятся все более заметными:

- Замедление разработки и тестирования – по мере увеличения объема кода работа над системой замедляется, усложняя процессы внесения изменений и тестирования.
- Ограниченная масштабируемость – отдельные компоненты приложения сложно или вовсе невозможно масштабировать независимо друг от друга, что ограничивает гибкость системы.
- Сложности при внедрении новых технологий – любые изменения в инфраструктуре или используемом языке разработки требуют значительных временных затрат из-за тесной связанности компонентов.
- Проблемы с развертыванием – даже небольшие изменения в коде требуют развертывания всего приложения, что может создавать перебои в работе [11].
- По мере того, как монолитная кодовая база продолжает разрастаться, доменная логика приложения становится запутанной и сложной для восприятия.

Это значительно затрудняет адаптацию новых разработчиков и ведет к повышению вероятности ошибок. На рисунке 2 представлена схема, демонстрирующая сложность монолита на примере домена биллинговой системы.

Практически все основные сервисы экосистемы демонстрируют признаки и проблемы, характерные для монолитной архитектуры. Часто изменение в одном сервисе требует соответствующих правок в других, а полноценная работа некоторых компонентов невозможна без запуска определенных взаимосвязанных сервисов. Например, Сервис работы с должниками (СРД) не может рассчитать сумму задолженности без доступа к

биллинговой системе, где выполняется расчет, даже несмотря на наличие у СРД данных о финансовом положении должника.

Эти зависимости указывают на то, что архитектура экосистемы ООО «Квартплата 24» подходит под определение «распределенный монолит». Несмотря на то, что сервисы развернуты в отдельных контейнерах и на разных серверах, между ними существует очень тесная связь, что затрудняет их независимую работу и масштабирование. Таким образом, проведен анализ основных сервисов экосистемы ООО «Квартплата 24» и подробно рассмотрены их характеристики и архитектурные особенности.

Рисунок 2 наглядно иллюстрирует степень сложности и взаимозависимости компонентов в рамках монолитной архитектуры биллинговой системы, демонстрируя проблему распределённого монолита, которая затрудняет масштабирование и независимую разработку сервисов.

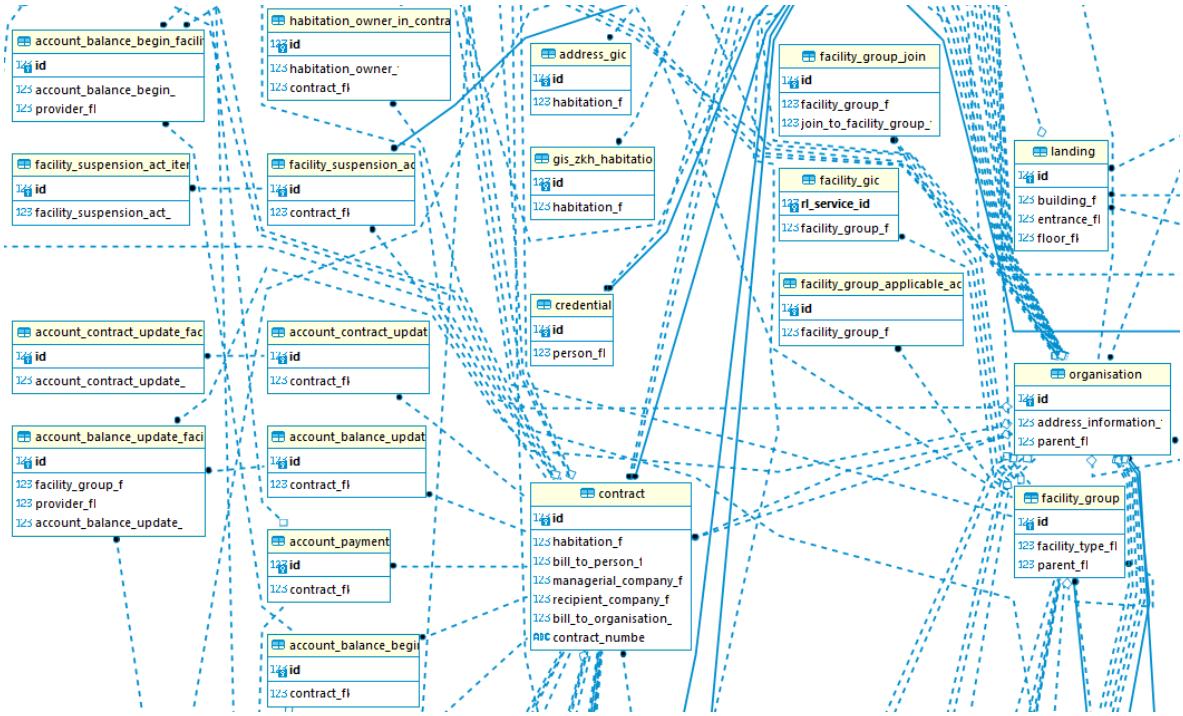

Рисунок 2 – Схема запутанного монолита на примере домена биллинговой системы

Современные распределенные вычислительные системы требуют эффективных методов взаимодействия между различными компонентами. В условиях растущей нагрузки, увеличения объема данных и необходимости обеспечения высокой производительности, унифицированный доступ к этим системам становится критическим фактором. Основной целью унифицированного интерфейса является создание единой точки взаимодействия, которая может обеспечить интеграцию разнородных систем и унифицировать методы доступа к данным. В рамках исследования проанализированы существующие подходы и технологии, применяемые для унифицированного доступа, с акцентом на использование реактивных архитектур и современных протоколов, таких как gRPC.

В рамках анализа существующих архитектурных решений основное внимание уделяется их способности обрабатывать большие объемы гетерогенных задач. Например, данные о времени отклика и загрузке системы используются для построения моделей прогнозирования и оптимизации работы системы. Сравнение технологий проводится на основе таких параметров, как масштабируемость, устойчивость к сбоям и скорость обработки данных. Это позволяет выбрать архитектуру, которая наилучшим образом соответствует требованиям распределенных вычислительных сред

Унифицированный доступ обеспечивает целостное представление данных и сервисов, что облегчает работу с распределенной системой и повышает ее производительность. На практике это достигается путем создания программного интерфейса, который позволяет абстрагировать пользователя от конкретных особенностей данных и методов их хранения.

Основные преимущества унифицированного доступа заключаются в:

– Снижении сложности интеграции различных компонентов системы за счет использования единого интерфейса, который скрывает детали реализации.

– Повышении производительности, так как унифицированный интерфейс позволяет оптимизировать маршрутизацию запросов и распределение нагрузки.
– Обеспечении отказоустойчивости и надежности за счет использования современных технологий асинхронной обработки и передачи данных.

Реактивная архитектура становится предпочтительной для создания унифицированных интерфейсов доступа к распределенным системам благодаря ряду преимуществ, таких как гибкость, отзывчивость, возможность масштабирования и устойчивость к ошибкам [3]. Реактивные системы работают на основе асинхронных неблокирующих вызовов, что позволяет им обрабатывать большое количество запросов, минимизируя время ожидания и задержки [1]. Это достигается благодаря:

– Асинхронной обработке данных, которая позволяет эффективнее распределять вычислительные ресурсы.
– Неблокирующим операциям, исключающим зависание системы при высоких нагрузках.
– Возможности масштабирования, позволяющей добавлять ресурсы в пиковые периоды нагрузки без значительных потерь производительности.

Реактивная архитектура особенно полезна в условиях гетерогенных систем, так как позволяет поддерживать адаптивность, автоматически подстраиваясь под изменения нагрузки и требования к системе.

Одним из способов избавления от анти-паттерна «распределенный монолит» является постепенный переход от монолитной архитектуры и ее разновидностей к микросервисной и реактивной архитектуре.

Системы, построенные в соответствии с принципами реактивной архитектуры, отличаются быстрой реакцией на запросы, высокой адаптивностью и возможностью простого масштабирования. Основой всех

этих преимуществ перед другими подходами является передача данных с помощью асинхронных неблокирующих сообщений.

На рисунке 3 представлены главные принципы реактивной архитектуры. Отзывчивость обеспечивается за счет использование асинхронных сообщений при передаче данных между компонентами информационной системы, что делает эти компоненты слабосвязанными. Такой подход позволяет управлять нагрузкой, осуществлять мониторинг очереди сообщений и предотвращать перегрузку получателя данных их поставщиком.

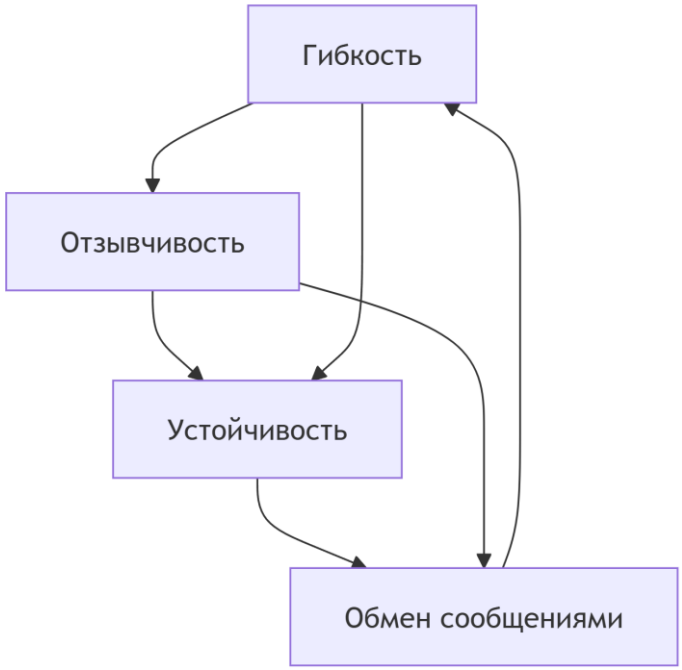

Рисунок 3 – Схема, демонстрирующая основные принципы реактивных систем

Гибкость системы достигается за счет динамического масштабирования при увеличении нагрузки. Например, в период расчета многократно возрастает нагрузка на сервис, отвечающий за проведение самого расчета, поэтому, помимо двух существующих экземпляров сервиса будут подняты еще два, и между ними уже будет распределяться нагрузка [16][17].

Устойчивость к ошибкам обеспечивается за счет использования паттерна Circuit Breaker, переотправку запросов и других подходов.

Следование принципам реактивной архитектуры позволит создавать системы, быстро реагирующие на запросы пользователей [47]. Помимо улучшения клиентского опыта взаимодействия с приложениями, повышения его уровня доверия к продукту, облегчается и взаимодействие с ними разработчиков за счет простоты определения и устранения ошибок. Все это способствует быстрому выходу продукта на рынок программного обеспечения по сравнению с конкурентами, использующих другие подходы проектирования.

Использование протокола gRPC представляет собой современное решение для удаленного вызова процедур (RPC), поддерживающее двустороннюю передачу данных с использованием HTTP/2 и Protocol Buffers [41]. Это значительно повышает скорость обмена данными между сервисами, снижая задержки и минимизируя накладные расходы. Применение gRPC вместо проприетарного протокола RAP, построенного на Akka I/O и TCP с Java-сериализацией, обеспечивает стабильную и высокопроизводительную работу распределенной системы, особенно в условиях растущей клиентской базы и увеличивающихся объемов данных.

RAP-сервис (Remote Access Point) в экосистеме используется для связи между клиентом и сервером, обеспечивая взаимодействие таких компонентов, как биллинговая система и сервис работы с должниками. RAP выполняет роль балансировщика нагрузки, но с ростом объемов данных производительность сервиса снижается, что приводит к заметному замедлению при передаче данных и выполнении различных операций [45]. Это вызывает неудовлетворенность у пользователей и указывает на необходимость поиска альтернативы, такой как gRPC.

Для оценки производительности gRPC и RAP был проведен ряд тестов. Тестирование проводилось с использованием "контракта" – виртуального договора с данными о лицевом счете, который представляет собой центральную сущность в системе. Основной целью тестирования стало измерение времени выполнения операций записи и чтения данных, включая

gRPC Streams для двусторонней передачи сообщений в рамках одного соединения.

Алгоритм тестирования включал:

– Отправку запроса от клиента с фиксацией времени в csv-файл.
– Генерацию массива данных на сервере с различными размерами (10, 50, 100, 250, 500, 1000, 2500 элементов).
– Отправку 200 сообщений с массивами данных обратно на клиент.
– Запись времени получения ответов на стороне клиента в csv-файл.

На основе полученных данных были построены графики, демонстрирующие время, затраченное на передачу данных и обработку запросов. Также сравнивались показатели при прямом доступе к системе и при удаленном доступе через VPN.

На рисунке 4 показана линейная зависимость времени отклика сервера от объёма обрабатываемых лицевых счетов: при увеличении нагрузки с 120 до 200 запросов в секунду время отклика растёт от 145 мс до 270 мс.

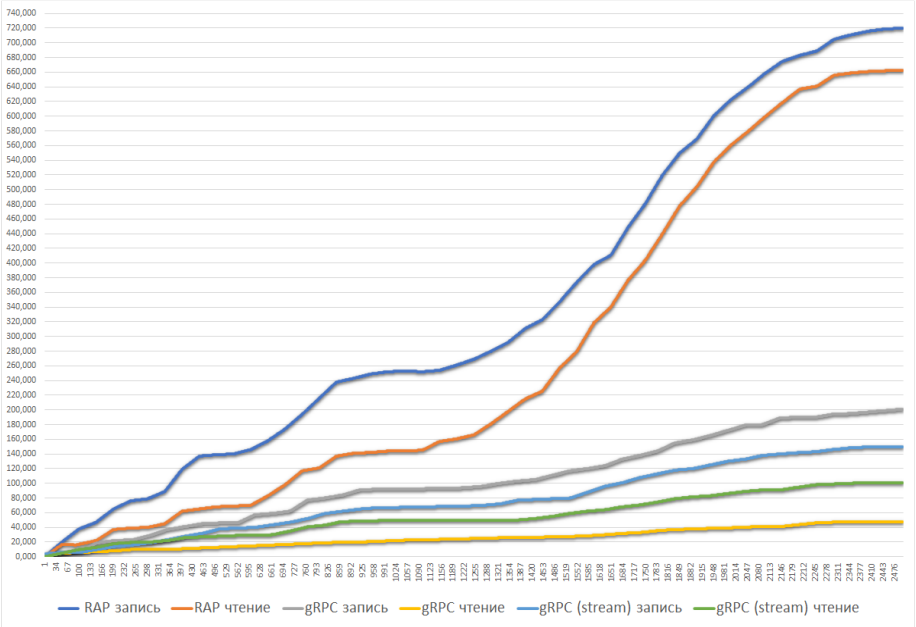

Рисунок 4 – График, демонстрирующий отношение кол-ва л/с и времени, затраченного на их обработку сервером

Такая прямо пропорциональная динамика подтверждает адекватность выбранной модели регрессии и подчёркивает необходимость своевременного масштабирования системы при прогнозируемом росте нагрузки, чтобы избежать достижения критических значений задержек.

Рисунок 5 иллюстрирует логику инициирования масштабирования и оповещений при превышении пороговых значений вероятности отказа компонентов системы.

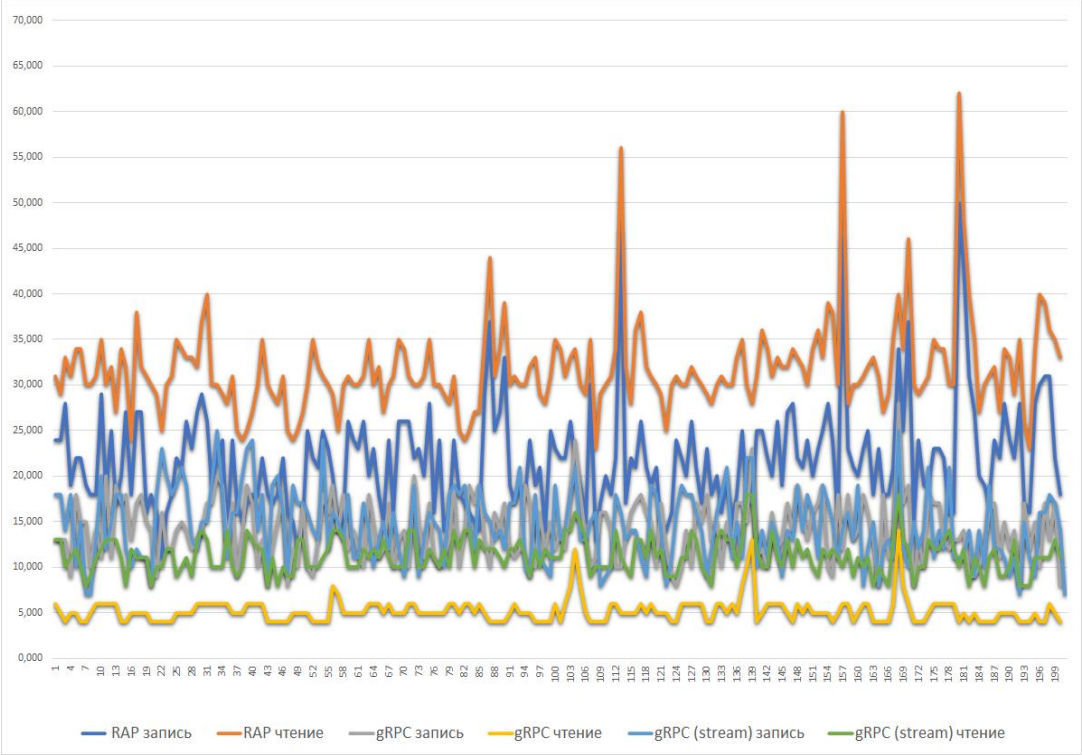

Рисунок 5 – График, отражающий кол-во времени, затрачиваемого на выполнение каждого запроса для массива размером 100 л/с

На схеме показаны ключевые метрики для микросервиса A и базы данных – загрузка CPU, количество активных соединений и частота ошибок – и критические уровни, при которых расчетная вероятность отказа становится высокой. Как только эта вероятность ($P > 0.7$) достигает заданного порога, система автоматически запускает процедуры масштабирования (добавление реплик уязвимого сервиса) и отправляет уведомление DevOps-команде для оперативного реагирования. Среднее время выполнения операций через gRPC

в четыре раза превышало показатели RAP. RAP показал значительные колебания времени обработки – различие между самым быстрым и самым медленным запросом составило до 2-3 раз, что указывает на низкую стабильность RAP.

Рисунок 6 демонстрирует зависимость времени выполнения каждого из семи тестов с разными объёмами массивов лицевых счетов для протоколов RAP и gRPC.

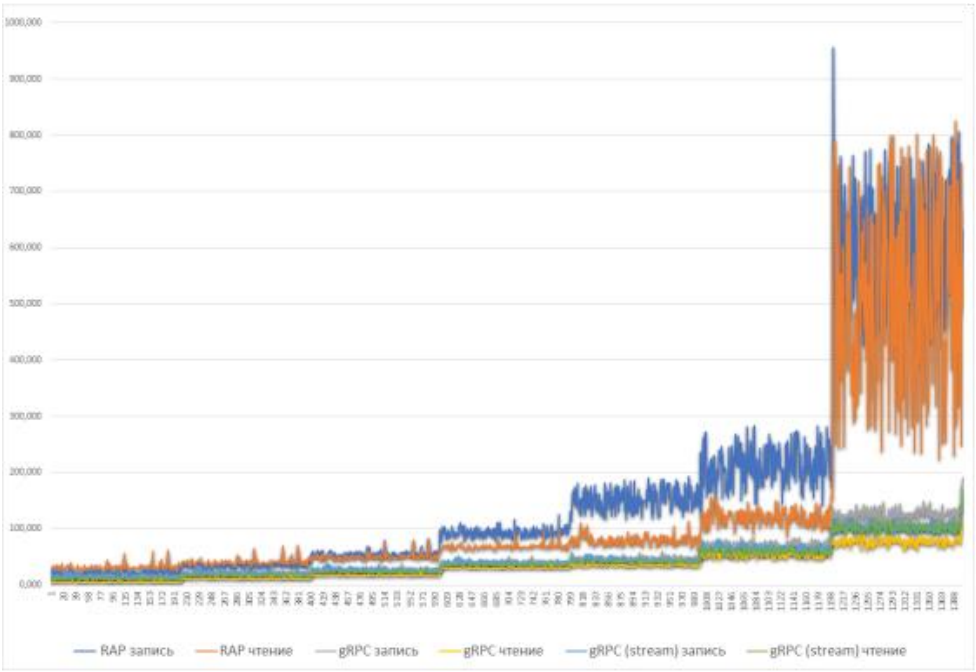

Рисунок 6 – График, демонстрирующий время по каждому запросу для всех семи тестов

На графике видно, что при увеличении размера запроса RAP показывает значительную вариативность: разброс времени обработки между самым быстрым и самым медленным запросом достигает 2–3-кратного значения. В то же время gRPC сохраняет высокую стабильность – время выполнения практически не изменяется при росте объёма данных, что подтверждает его предсказуемость и эффективность в условиях нагрузочного тестирования

Рисунок 7 иллюстрирует результаты тестирования протоколов RAP и gRPC в условиях удалённого доступа через VPN, приближённого к реальному

окружению. По оси X отложен размер массива лицевых счетов (л/с), по оси Y – время обработки запроса.

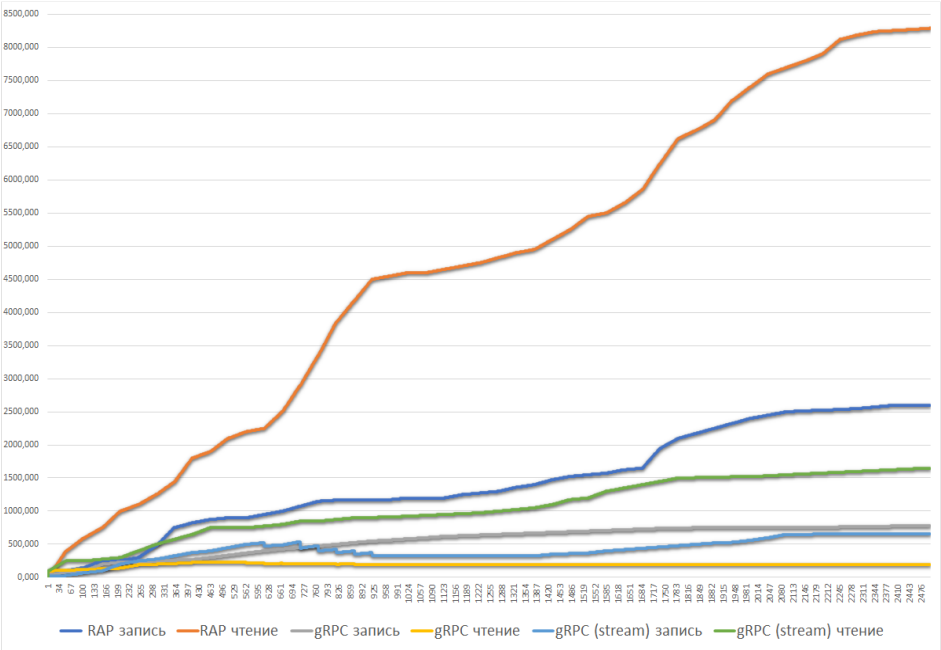

Рисунок 7 – График, демонстрирующий отношение кол-ва л/с ко времени, затрачиваемого на его обработку в приближенных к реальным условиям

На графике видно, что даже при существенном снижении пропускной способности сети gRPC демонстрирует практически линейный и стабильный рост времени передачи (не превышая ~1 с при 2 500 л/с), тогда как RAP испытывает экспоненциальное увеличение задержек, достигая до 8,5 с при максимальной нагрузке. Эти результаты подчёркивают высокую устойчивость и предсказуемость gRPC в распределённых системах с нестабильными сетевыми условиями и подтверждают необходимость его использования для обеспечения надёжного обмена данными в удалённом доступе.

Проведенное тестирование выявило существенные недостатки текущего протокола RAP, особенно в условиях увеличивающейся нагрузки. gRPC продемонстрировал себя как более стабильный и высокопроизводительный вариант для клиент-серверного взаимодействия в распределенной системе.

Результаты исследования подчеркивают необходимость перехода на gRPC для обеспечения надежной и эффективной работы экосистемы.

Рисунок 8 демонстрирует суммарное время выполнения всех семи нагрузочных тестов с различными объёмами массивов лицевых счетов при удалённом доступе через VPN.

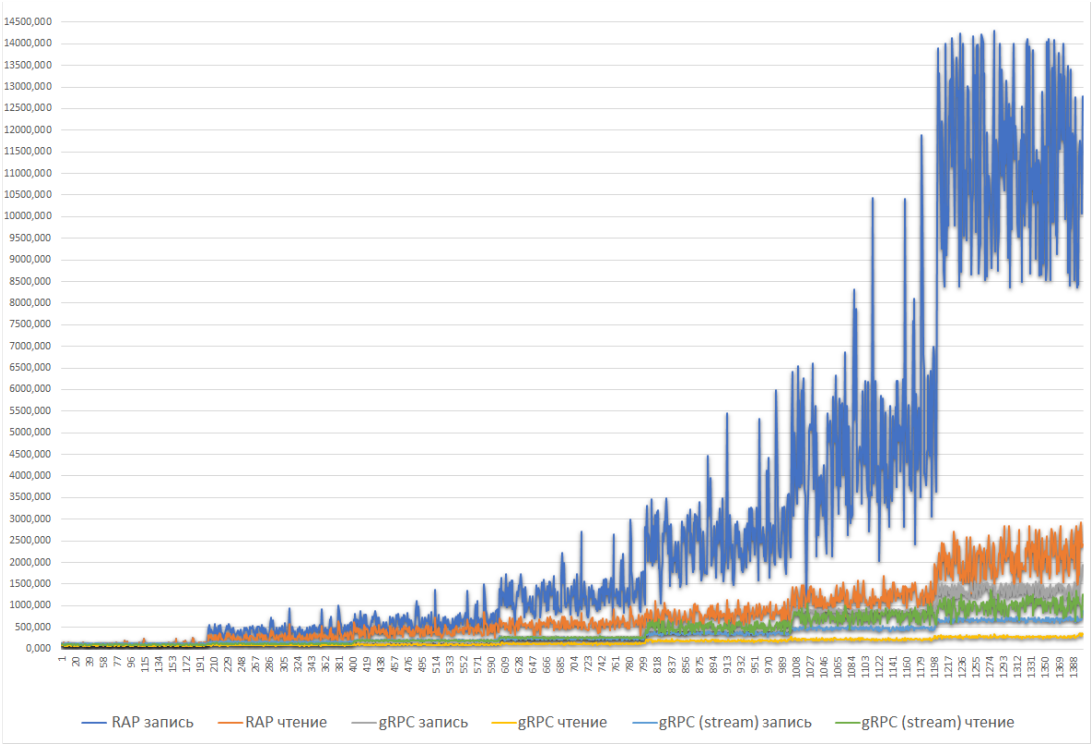

Рисунок 8 – График, отражающий время, затраченное на выполнение каждого запроса для всех семи тестов с использованием удаленного доступа

На графике видно, что gRPC обеспечивает практически постоянное время обработки – не превышающее одной секунды даже при максимальной нагрузке – тогда как RAP показывает значительную нестабильность: время отклика варьируется, и разница между двумя идентичными запросами достигает 2–4-кратного значения.

Разработка архитектуры API Gateway основывается на анализе статистических данных, включая объемы запросов, пиковые нагрузки и распределение ресурсов между компонентами системы [18]. Эти данные позволяют эффективно распределять запросы, балансировать нагрузку и

предотвращать сбои в работе системы. Применение методов анализа данных в процессе проектирования обеспечивает адаптивность и стабильность системы даже при увеличении нагрузки API Gateway является еще одним важным компонентом унифицированного доступа к распределенной системе. Это промежуточный слой между клиентами и микросервисами, обеспечивающий единую точку входа и управления для всех запросов, что упрощает управление доступом к микросервисам и способствует повышению безопасности. Основные функции API Gateway включают:

- Маршрутизацию запросов, направляя их на нужные микросервисы в зависимости от типа запроса и его параметров.
- Агрегацию ответов, позволяющую собирать результаты из нескольких сервисов и возвращать их в одном ответе клиенту.
- Управление аутентификацией и авторизацией, обеспечивая централизованную проверку доступа к ресурсам.
- Выполнение функций мониторинга и логирования, что упрощает отслеживание производительности и выявление узких мест в системе.

В ходе исследования для реализации API Gateway был выбран Envoy Proxy, показавший высокую производительность в сравнении с конкурентами, такими как Nginx и Kong. Envoy Proxy обеспечивает поддержку gRPC без дополнительных расширений и позволяет работать с JWT-токенами для аутентификации, что делает его оптимальным выбором для унифицированного интерфейса [5].

В ходе анализа существующих решений и технологий были рассмотрены ключевые подходы и инструменты для организации унифицированного доступа к распределенным системам. В частности, использование таких архитектурных компонентов, как API Gateway, Envoy Proxy, и современных протоколов взаимодействия (например, gRPC), позволило выделить наиболее эффективные методы организации взаимодействия между сервисами [24][25]. Эти технологии создают надежную

основу для проектирования унифицированного интерфейса, позволяющего интегрировать различные компоненты системы и обеспечить их эффективное взаимодействие.

Однако с увеличением объемов данных и ростом количества пользователей возникает необходимость оптимизации текущей архитектуры. Проблемы, такие как растущее время отклика и сложности масштабирования, указывают на ограниченность традиционных монолитных подходов и подчеркивают важность разработки более гибкой и адаптивной архитектуры. Оптимизация унифицированного интерфейса позволит не только ускорить обмен данными между сервисами, но и улучшить управляемость и устойчивость всей системы.

Таким образом, рассмотренные в данном разделе технологии и архитектурные решения будут служить основой для следующего этапа – проектирования и оптимизации унифицированного интерфейса доступа [13]. Этот этап предполагает создание гибкой и масштабируемой архитектуры, которая сможет эффективно поддерживать распределенные вычислительные среды и удовлетворять потребности пользователей в условиях высокой нагрузки.

1.3 Методы сбора и обработки данных в распределённых системах

В распределённых вычислительных системах, таких как экосистема сервисов компании «Квартплата 24», сбор и обработка данных играют ключевую роль в обеспечении оптимальной производительности и устойчивости архитектуры. Эти данные позволяют принимать обоснованные решения, направленные на улучшение маршрутизации запросов, балансировки нагрузки и повышения отказоустойчивости системы.

Сбор данных осуществляется из нескольких источников, которые отражают текущее состояние системы и её компонентов. Одним из центральных инструментов мониторинга является Prometheus, который

используется для автоматического сбора метрик, таких как время отклика, загрузка процессора и частота ошибок [48]. Эти метрики предоставляют объективное представление о работе системы и позволяют выявить узкие места в её архитектуре. Например, в рамках проектирования API Gateway, описанного в разделе 2.1, метрики времени отклика и нагрузки помогли определить основные проблемы в маршрутизации запросов. Собранные данные также используются для анализа динамики изменения производительности при увеличении нагрузки, что позволяет оперативно вносить изменения в конфигурацию системы. Для мониторинга системы используются инструменты, которые автоматизируют сбор данных и обеспечивают их хранение в удобном для анализа формате. Одним из таких инструментов является Prometheus, применяемый для сбора метрик производительности, включая время отклика, загрузку CPU и частоту ошибок.

Обработка и анализ эксплуатационных данных начинается с очистки сырого потока метрик и логов от дубликатов, пропусков и некорректных записей, после чего все события агрегируются в ключевые показатели – например, среднее время отклика или общую загрузку CPU. На основании этих обобщённых метрик в среде Apache Spark выполняются кластеризация и регрессионное моделирование, что позволяет не только выявлять текущие узкие места в работе API Gateway и микросервисов, но и прогнозировать пиковые нагрузки с последующей корректировкой баланса запросов [28]. Полученные результаты визуализируются в Grafana-дашбордах, дающих операторам систему мониторинга в реальном времени и автоматизированные оповещения при превышении критических порогов.

Интеграция данных из Prometheus-метрик и логов приложений обеспечивает целостное представление о поведении распределённой системы: сведения о времени отклика, ошибках и использовании ресурсов сводятся в единую картину, на основе которой принимаются как оперативные, так и стратегические решения. Такой конвейер обработки – от очистки и агрегирования до прогнозирования и визуализации – гарантирует

устойчивость и масштабируемость архитектуры при росте нагрузки и новых эксплуатационных требованиях [20]. Для систематизации подходов, используемых в данной работе для сбора и обработки данных, представлена таблица 1, которая отражает методы, их описание, примеры применения и используемые инструменты.

Таблица 1 – Методы сбора и обработки данных

Метод	Описание	Применение	Инструменты
Сбор метрик производительности	Автоматический сбор данных о времени отклика, загрузке CPU, частоте ошибок.	Анализ производительности API Gateway, выявление узких мест.	Prometheus, Grafana
Анализ логов	Извлечение данных о запросах, ошибках и временных паттернах.	Оптимизация взаимодействия между сервисами, анализ работы базы данных.	Лог-файлы приложений, Java
Очистка данных	Удаление дубликатов, устранение некорректных данных.	Подготовка данных для анализа и визуализации.	Скрипты на Java и Scala
Агрегация данных	Объединение данных для получения обобщённых метрик	Построение дашбордов для мониторинга в реальном времени.	Prometheus, Java
Регрессионный анализ	Прогнозирование параметров системы на основе исторических данных.	Оценка влияния изменений в архитектуре API Gateway, прогнозирование нагрузки.	Scala (Spark MLlib), Java
Кластеризация задач	Группировка задач по схожим характеристикам для эффективного	Оптимизация маршрутизации запросов и повышения пропускной способности.	Scala (Spark MLlib), Java

	распределения нагрузки.		
Визуализация данных	Создание дашбордов для анализа текущих метрик и выявления аномалий.	Мониторинг в реальном времени	Grafana

Таблица 1 демонстрирует, что использование Java и Scala обеспечивает гибкость в реализации процессов обработки данных, включая анализ метрик, кластеризацию задач и прогнозирование производительности. Интеграция этих методов позволяет эффективно решать задачи оптимизации архитектуры распределённых вычислительных систем.

Методы сбора и обработки данных могут быть дополнены использованием алгоритмов машинного обучения. Это позволит автоматизировать анализ логов и прогнозировать возможные сбои на основе данных [21]. Кроме того, более глубокая интеграция с инструментами управления метриками, такими как Grafana, позволит оперативно реагировать на изменения в работе системы. На рисунке 9 наглядно показано, как данные проходят через весь процесс – от сбора до визуализации, что обеспечивает основу для проектирования архитектурных решений и их оптимизации.

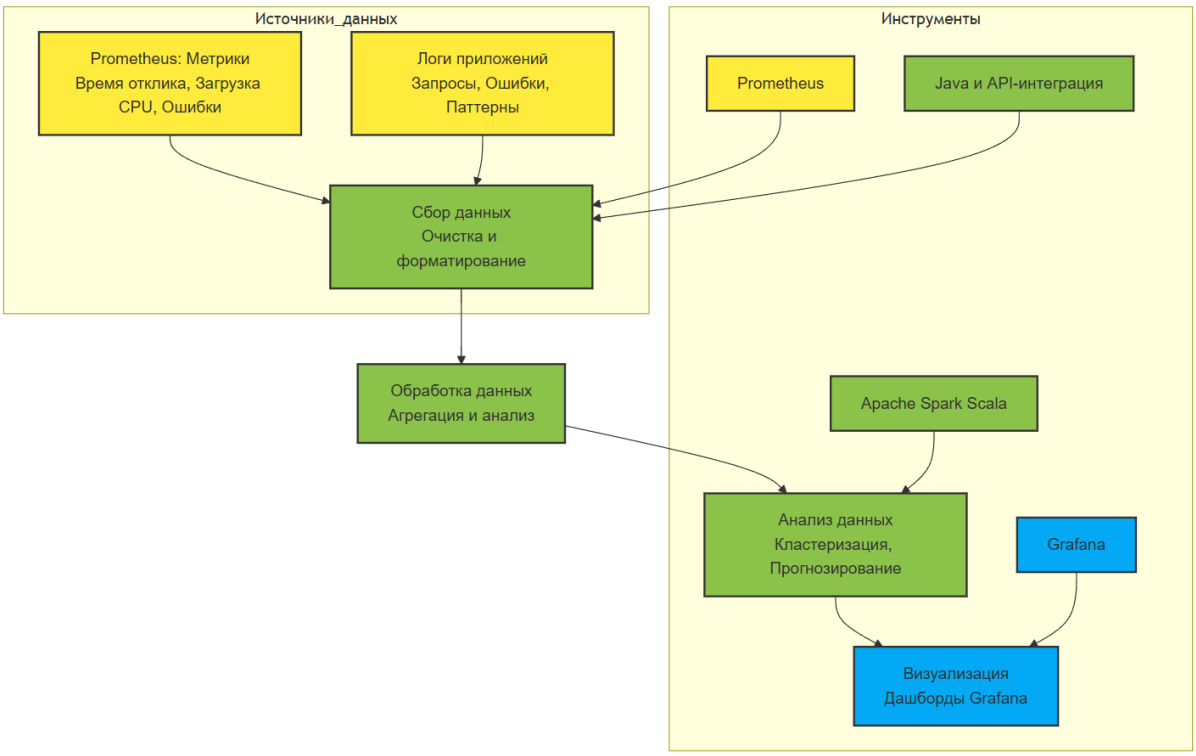

Рисунок 9 - Процесс сбора, обработки и использования данных

В контексте компании «Квартплата 24» эти методы стали основой для проектирования и реализации унифицированного интерфейса доступа, а также для улучшения взаимодействия между микросервисами.

Выводы по разделу 1

Анализ существующих решений и технологий показал, что эффективность распределённых систем напрямую зависит от архитектуры и глубины аналитики эксплуатационных данных. В экосистеме компании «Квартплата 24» выявлены ограничения монолитной архитектуры, проявляющиеся в слабой масштабируемости и высокой связности сервисов. Это подтверждает необходимость перехода к более гибкой и устойчивой микросервисной модели.

Особую роль играет анализ логов, метрик и профилирования. Использование Prometheus, Grafana и инструментов логирования обеспечивает раннее выявление проблем и позволяет прогнозировать поведение системы. Внедрение реактивной архитектуры и переход на gRPC значительно снижает время отклика и повышает стабильность межсервисного взаимодействия.

API Gateway на базе Envoy централизует управление доступом, маршрутизацией и мониторингом, повышая отказоустойчивость и упрощая поддержку. Интеграция методов анализа данных, таких как кластеризация и регрессия, позволяет обоснованно принимать архитектурные решения, выявлять узкие места и адаптировать систему к росту нагрузки.

Таким образом, раздел обосновывает необходимость перехода к адаптивной архитектуре, в которой данные и аналитика играют ключевую роль в обеспечении надёжности и производительности распределённых систем.

2 Проектирование и оптимизация унифицированного интерфейса доступа в распределенных системах

2.1 Проектирование архитектуры API Gateway на основе анализа данных

Ранее разработанное решение Remote Access Point (RAP) столкнулось с рядом критических проблем, включая низкую производительность и недостаточную стабильность, что негативно влияло на отказоустойчивость и отзывчивость системы. Кроме того, реализация API-шлюза в RAP оказалась неполной – шлюз обрабатывал запросы только от некоторых приложений, в то время как другие напрямую взаимодействовали с клиентами. Это привело к запутанности архитектуры и затруднениям в поддержке кодовой базы, так как разные API взаимодействовали между собой неоднородно и без единой системы. Кроме того, код RAP был сложен для понимания из-за отсутствия документации и использования устаревших технологий, что затрудняло любые изменения в его работе.

Проектируемый в рамках этой работы API Gateway призван решить указанные недостатки, обеспечивая высокую отказоустойчивость, полную реализацию паттерна API-шлюза, документированность и применение современных технологий. С учетом того, что экосистема состоит из множества независимых сервисов, работающих в отдельных процессах и использующих разные протоколы передачи данных, новый шлюз должен централизовать и упростить их взаимодействие [26].

Основные функции API Gateway:

– Маршрутизация запросов от клиентов к микросервисам – API Gateway должен уметь перенаправлять запросы к нужным микросервисам, обеспечивая единую точку доступа для всех клиентских запросов.

– Агрегация ответов от микросервисов – в случае необходимости, шлюз объединяет ответы от нескольких микросервисов в единый ответ для клиента, упрощая взаимодействие и повышая удобство использования системы.
– Аутентификация и авторизация пользователей – API Gateway обеспечивает обработку запросов на доступ к защищенным ресурсам, управляя аутентификацией и выдачей временных токенов.
– Реализация дополнительных функций – шлюз также берет на себя задачи логирования, кэширования, ограничения скорости и мониторинга, которые критичны для эффективной работы всей системы.
– Работа с разными протоколами передачи данных – поддержка как минимум HTTP и gRPC позволяет использовать API Gateway для взаимодействия сервисов с разными требованиями к передаче данных.

Преимущества использования API Gateway. API Gateway служит промежуточным слоем между клиентами и микросервисами, предоставляя единый интерфейс для всех взаимодействий с экосистемой [27].

Это позволяет:

– Упростить управление доступом к микросервисам, создавая централизованную точку входа для всех запросов.
– Обеспечить гибкость и расширяемость за счет поддержки различных протоколов и механизмов маршрутизации.
– Повысить надежность и отказоустойчивость за счет балансировки нагрузки и возможности добавления новых экземпляров сервисов в production-среде.

На рисунке 10 представлена схема взаимодействия клиентов с различными интерфейсами приложения через API Gateway.

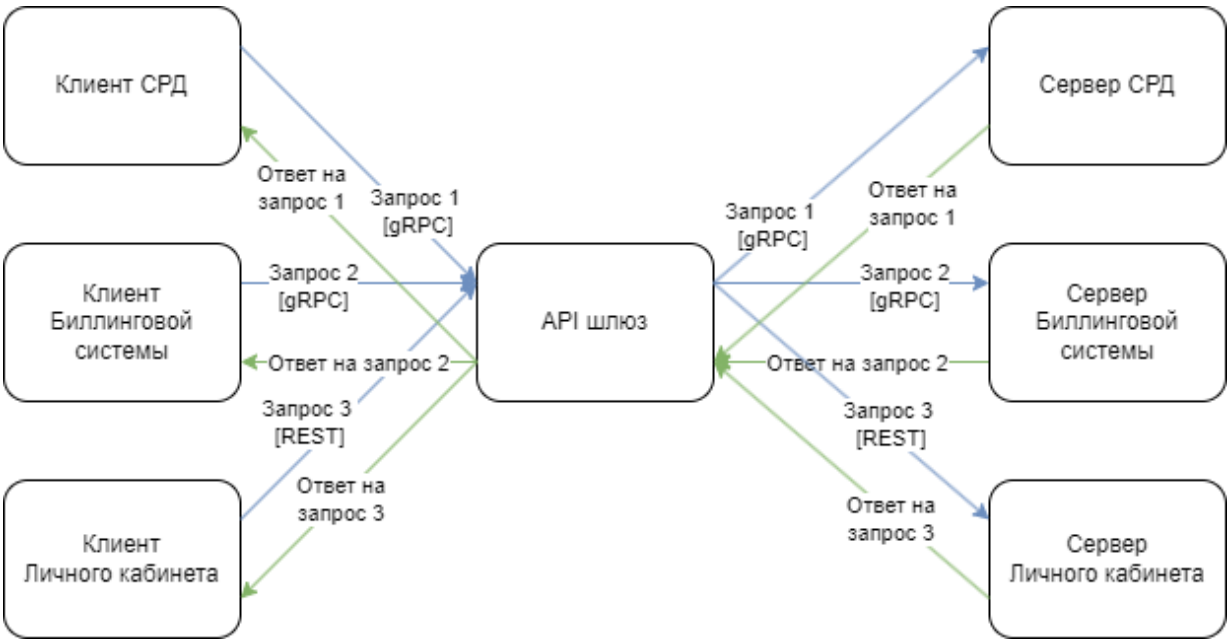

Рисунок 10 – Схема взаимодействия клиента с интерфейсами приложения с помощью API Gateway

С помощью API-шлюза внутренние системы организации изолированы от пользователей, что позволяет централизованно управлять взаимодействием всех подсистем, упрощая как работу клиентов, так и поддержку самих сервисов. Разработка и внедрение API Gateway решают проблему сложности взаимодействия между микросервисами и обеспечивают надежную, производительную и гибкую архитектуру, способную адаптироваться к изменяющимся требованиям и увеличению нагрузки.

2.2 Инструменты и технологии для реализации унифицированного интерфейса доступа

Для успешного создания API Gateway требовалось выбрать инструменты, которые соответствуют специфическим требованиям реактивной архитектуры и поддерживают высокую производительность, надежность и гибкость. Сегодня рынок предлагает множество инструментов для построения API Gateway, поэтому важным этапом стало определение

подходящего набора технологий, которые обеспечат эффективную маршрутизацию, балансировку нагрузки и безопасность.

Для взаимодействия через протокол gRPC была выбрана библиотека Akka gRPC. Эта библиотека обеспечивает поддержку потокового взаимодействия серверов и клиентов поверх Akka Streams и Akka HTTP. Она также включает генератор, который на основе протоколов, описанных в protobuf, создает модельные классы, API службы, а также код сервера и клиента. Это облегчает процесс интеграции gRPC в унифицированный интерфейс доступа и упрощает создание масштабируемых и надежных сервисов.

Ключевым требованием при выборе прокси-сервера для API Gateway было его способность работать с протоколом gRPC без необходимости использования дополнительных плагинов или расширений. Были рассмотрены три основных решения:

- Nginx – высокопроизводительный веб-сервер и обратный прокси, часто используемый в качестве API-шлюза для серверных приложений. Однако его поддержка gRPC ограничена и требует использования дополнительных модулей.
- Envoy – современный прокси, разработанный для сетевого взаимодействия между микросервисами. Он поддерживает L7 управление трафиком, работает с gRPC и может выступать в роли API-шлюза, что делает его универсальным инструментом для распределенных систем.
- Kong – масштабируемый API-шлюз, работающий на базе Nginx и Lua, поддерживающий добавление новых функций через плагины. Его гибкость и возможность настройки делают его отличным выбором для управления трафиком.

На основе результатов сравнительного анализа (рисунок 11 и рисунок 12), где было отмечено, что Envoy значительно превосходит другие решения

по производительности при работе с HTTP и HTTPS, он был выбран для реализации API Gateway.

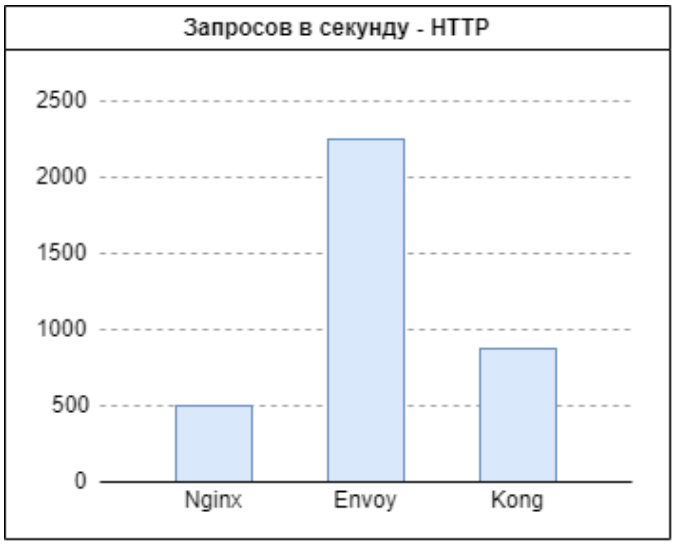

Рисунок 11 – Диаграмма, отражающая кол-во запросов в секунду по протоколу HTTP

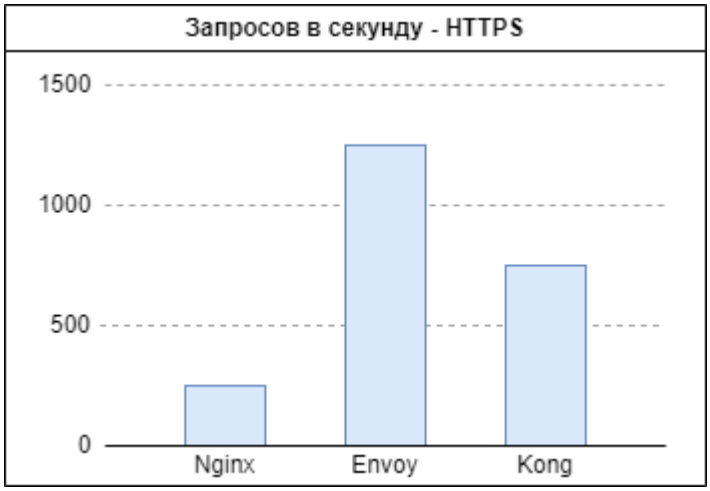

Рисунок 12 – Диаграмма, отражающая кол-во запросов в секунду по протоколу HTTPS

Диаграмма на рисунке 11 показывает, как каждый из трех прокси-серверов обрабатывает запросы по протоколу HTTP. Визуально можно отметить, что Nginx обрабатывает наименьшее количество запросов в секунду среди представленных серверов. Хотя Nginx является проверенным решением

для веб-серверов и обратного проксирования, его производительность по протоколу HTTP в этом тесте оказалась ниже, чем у остальных инструментов.

Диаграмма на рисунке 12 иллюстрирует количество запросов в секунду, которые могут обработать Nginx, Envoy и Kong при работе с зашифрованным трафиком по протоколу HTTPS. Результаты показывают следующую картину, Nginx снова обрабатывает меньше всего запросов в секунду по сравнению с Envoy и Kong, что может быть связано с дополнительной нагрузкой на обработку шифрования, не оптимизированной для высоких объемов HTTPS-трафика в этом контексте.

Envoy демонстрирует наивысшую производительность среди всех протестированных серверов, обрабатывая значительно больше запросов в секунду, чем Nginx и Kong. Это связано с его оптимизацией для работы в распределенных микросервисных средах и поддержкой сетевых взаимодействий на уровне L7. Envoy специально разработан для высокого уровня нагрузки и предлагает расширенные возможности маршрутизации и управления трафиком.

Kong занимает среднюю позицию по количеству запросов в секунду, превосходя Nginx, но уступая Envoy. Kong, работающий на базе Nginx, дополнен возможностью использования Lua-плагинов для гибкой настройки и расширения функционала, однако, по сравнению с Envoy, он менее оптимизирован для обработки большого количества запросов в секунду. Из этой диаграммы можно сделать вывод, что Envoy является наиболее производительным решением при работе с запросами по протоколу HTTP, что делает его подходящим выбором для создания API Gateway в высоконагруженной системе, ориентированной на HTTP-трафик [32].

Envoy продолжает лидировать, демонстрируя наивысшую производительность даже при работе с HTTPS, что указывает на его оптимизацию для защищенных соединений и подтверждает его статус надежного прокси-сервера для API Gateway. Высокая производительность при работе с HTTPS также делает его предпочтительным выбором для систем, где

безопасность и скорость критически важны. Kong снова показывает производительность выше, чем у Nginx, но ниже, чем у Envoy. Несмотря на то, что Kong поддерживает HTTPS, его архитектура и функциональные плагины не позволяют ему достичь таких же показателей, как у Envoy. Эти результаты подтверждают, что Envoy превосходит другие решения по количеству обрабатываемых HTTPS-запросов, что особенно важно для API Gateway, работающего в условиях повышенных требований к безопасности и производительности [34].

На основании диаграмм на рисунках 11 и 12 можно сделать вывод, что Envoy является наиболее производительным прокси-сервером как для HTTP, так и для HTTPS-трафика. Его возможности по обработке большого количества запросов в секунду делают его лучшим выбором для высоконагруженной микросервисной системы. Это решение обеспечивает стабильность, скорость и гибкость, необходимые для API Gateway, и подходит для работы в современных распределенных системах, где важны и производительность, и безопасность.

В соответствии с требованиями к безопасности, API Gateway должен поддерживать аутентификацию пользователей с использованием временных токенов – JWT (JSON Web Token). JWT является открытым стандартом (RFC 7519) для создания токенов доступа на основе формата JSON. Токен создается на сервере, подписывается секретным ключом и передается клиенту, который использует его для подтверждения своей личности при доступе к ресурсам системы [35].

Для реализации авторизации с использованием JWT были рассмотрены два инструмента:

- OAuth 2.0 – стандарт авторизации, который позволяет предоставлять доступ к ресурсам, не раскрывая при этом данные пользователя.
- OpenID Connect – надстройка над OAuth 2.0, которая добавляет поддержку профиля пользователя и позволяет хранить информацию о пользователе для использования в нескольких сервисах.

Поскольку экосистема ООО «Квартплата 24» не нуждается в централизованном хранилище данных о пользователях, было решено использовать OAuth 2.0 без дополнительных функций OpenID Connect, что позволяет избежать избыточности и упрощает разработку.

Для хранения данных о выданных токенах и других сущностей потребовалась интеграция с реляционной базой данных. Были рассмотрены два фреймворка:

- Hibernate – фреймворк ORM, который позволяет отображать структуры реляционной базы данных на объекты в объектно-ориентированном языке программирования.
- Slick – фреймворк FRM, который использует функциональное реляционное отображение, оптимизирован для работы с языком Scala и поддерживает реактивное программирование.

Преимущества Slick, такие как типобезопасные запросы, поддержка асинхронного стиля программирования и гибкость при работе с чистым SQL и FRM, сделали его предпочтительным выбором. Это решение соответствует принципам реактивной архитектуры и позволяет создавать высокопроизводительные, надежные приложения.

В связи с тем, что другие сервисы экосистемы используют PostgreSQL для хранения данных, было решено использовать ту же СУБД для унификации и совместимости. PostgreSQL – объектно-реляционная система управления базами данных, которая отличается высокой производительностью, надежностью и расширяемостью, что идеально подходит для поддержания стабильной работы API Gateway.

Выбор и интеграция инструментов, таких как Akka gRPC, Envoy, OAuth 2.0, Slick и PostgreSQL, позволяют создать API Gateway, который соответствует требованиям к производительности, безопасности и гибкости. Это обеспечивает создание эффективной архитектуры, которая способна выдерживать высокие нагрузки и адаптироваться к потребностям растущей

системы, выполняя роль унифицированного интерфейса для распределенной экосистемы ООО «Квартплата 24».

2.3 Архитектура классов и база данных для управления доступом

Проектируемый унифицированный интерфейс доступа состоит из нескольких модулей, каждый из которых выполняет определенные функции по обработке входящих запросов и управлению данными. Основными методами взаимодействия являются REST-запросы и gRPC-вызовы, что позволяет гибко и эффективно работать с различными типами данных и системными вызовами [49]. На основе проектирования классов и структуры базы данных создается основа для реализации стабильной и надежной системы управления доступом.

В таблице 2 приведено краткое описание основных классов, задействованных в системе.

Таблица 2 – Описание существующих классов

Название класса	Роль класса
Launcher	Инициализирует запуск сервиса, включая настройку слушателей для REST и gRPC запросов.
Httprequestlistener	Слушает и обрабатывает входящие HTTP-запросы.
Rpcrequestlistener	Слушает и обрабатывает gRPC-вызовы.
Configparser	Загружает и обрабатывает конфигурационные данные для настройки сервиса.
Httprequesthandler	Обрабатывает непосредственно входящие HTTP-запросы.
Rpcrequesthandler	Обрабатывает непосредственно gRPC-вызовы.
Oauthprovider	Управляет аутентификацией, работая с токенами доступа для авторизации пользователей.
Servicerpcabstractmapper	Базовый класс, отвечающий за преобразование ответов HTTP в формат gRPC.
Debtservicerpcmapper	Преобразует ответы HTTP в gRPC для службы работы с задолженностями (СРД).
Ssbsservicerpcmapper	Преобразует ответы HTTP в gRPC для биллинговой системы.

На рисунке 13 представлена диаграмма классов, отражающая структуру разрабатываемого сервиса унифицированного доступа.

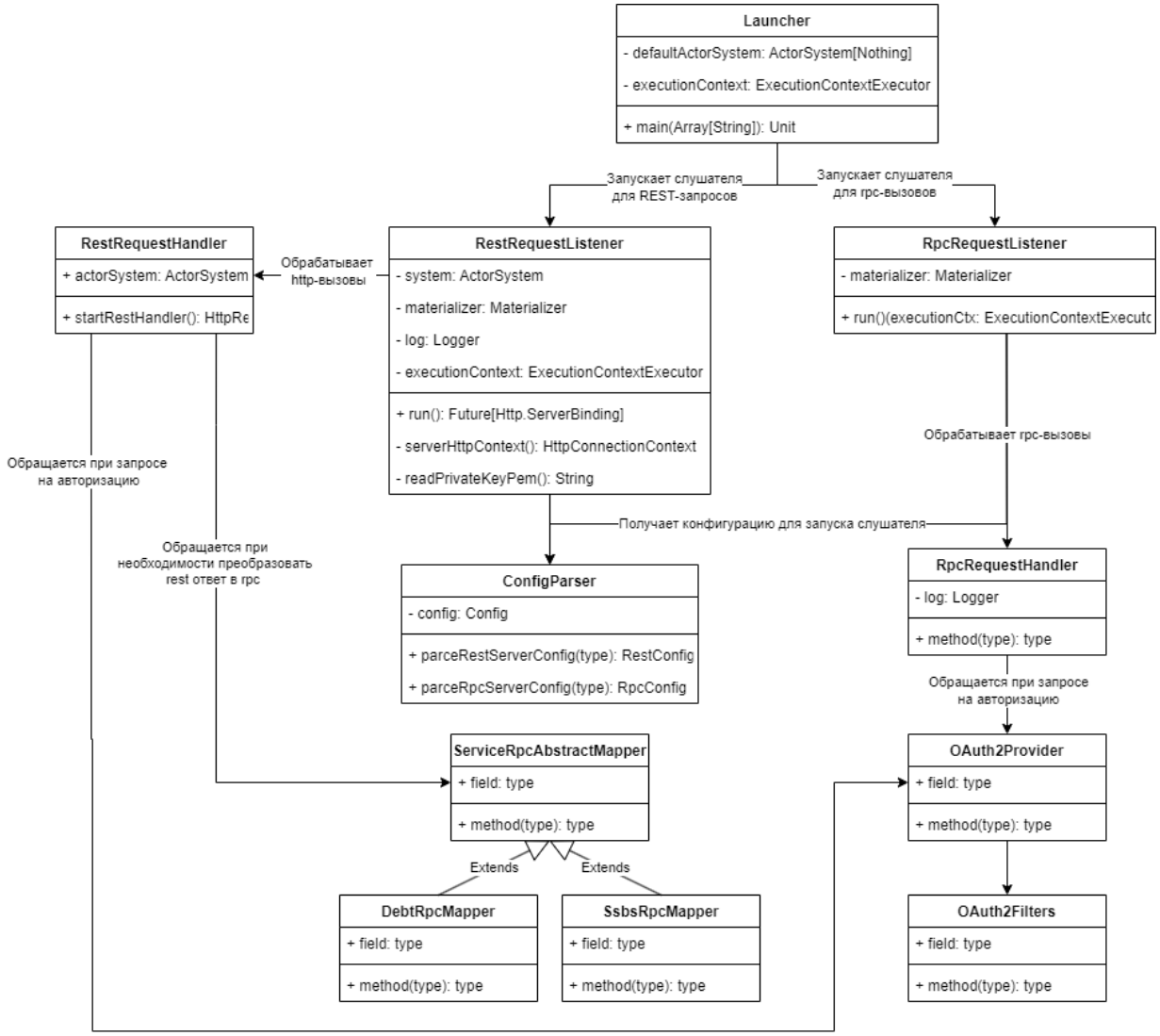

Рисунок 13 – Диаграмма классов интерфейса унифицированного доступа

Диаграмма показывает взаимосвязи между классами, их основные методы и поля. Основные компоненты включают:

- Launcher – центральный класс, ответственный за инициализацию сервиса и запуск слушателей для REST и gRPC запросов.
- RestRequestListener и RpcRequestListener – классы, управляющие входящими запросами, соответственно, для REST и gRPC, обрабатывая их с помощью соответствующих обработчиков запросов.

– ConfigParser – класс, который считывает конфигурации для запуска слушателей и обеспечивает корректную настройку компонентов сервиса.
– OAuth2Provider – реализует логику авторизации, обрабатывая запросы на авторизацию и управление токенами доступа.
– ServiceRpcAbstractMapper – абстрактный класс для преобразования HTTP ответов в формат gRPC. Классы DebtRpcMapper и SsbsRpcMapper наследуют его и обеспечивают преобразование специфичных ответов для соответствующих систем [36].

На рисунке 14 представлена ER-диаграмма базы данных, разработанной для хранения данных о выданных токенах доступа.

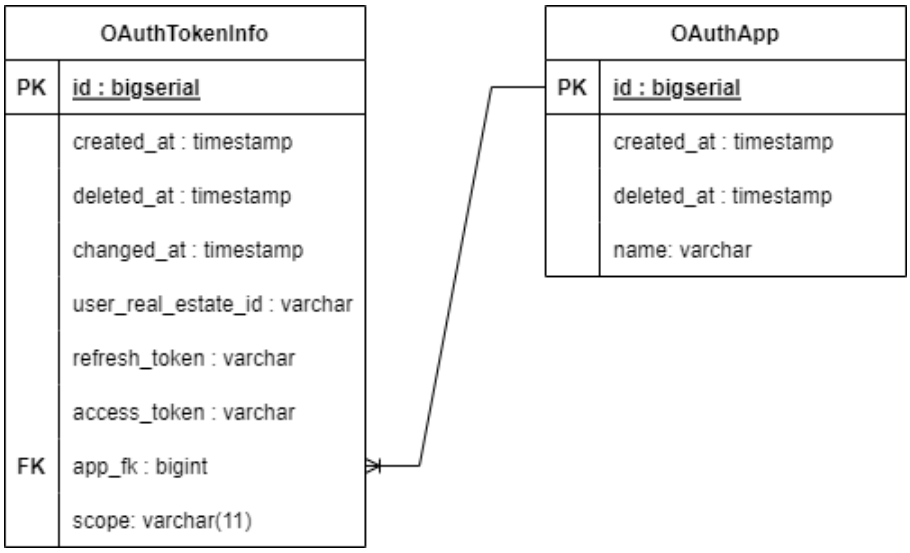

Рисунок 14 – ER-диаграмма базы данных для хранения токенов

База данных состоит из двух основных таблиц. OAuthTokenInfo – хранит данные о выданных токенах для пользователей. Включает следующие поля:

– created_at, deleted_at, changed_at – метки времени, отражающие создание, удаление и изменения записи.
– user_real_estate_id – уникальный идентификатор пользователя, который может быть связан с идентификатором в других системах.

– refresh_token и access_token – хранят данные для обновления и доступа, соответственно.
– app_fk – внешний ключ, связывающий запись с таблицей OAuthApp, что позволяет определить, для какого приложения выдан токен.

OAuthApp – хранит информацию о приложениях, которые используют OAuth для аутентификации пользователей. Включает следующие поля:

– name – название приложения.
– created_at и deleted_at – метки времени для отслеживания момента создания и удаления записи.

Связь между таблицами OAuthTokenInfo и OAuthApp реализована через внешний ключ app_fk и является отношением «многие к одному», что позволяет одному приложению иметь множество токенов [37].

Построение архитектуры классов и создание ER-диаграммы базы данных для хранения токенов обеспечивают гибкость, масштабируемость и безопасность системы. Диаграммы и описания позволяют структурировать работу над интерфейсом унифицированного доступа, обеспечивая эффективное управление авторизацией, маршрутизацией и поддержкой различных протоколов.

2.4 Интеграция API Gateway в архитектуру сервисов

Сейчас IT-экосистема компании «Квартплата 24» представляет собой гибридную архитектуру, в которой сочетаются как монолитные приложения, так и микросервисы. Эти приложения и сервисы работают в одном кластере и общаются между собой напрямую, что иногда усложняет поддержку и масштабирование системы. Разработка и интеграция нового API Gateway, представленного на рисунке 15, направлены на улучшение взаимодействия между сервисами и создание условий для более легкого перехода к микросервисной архитектуре.

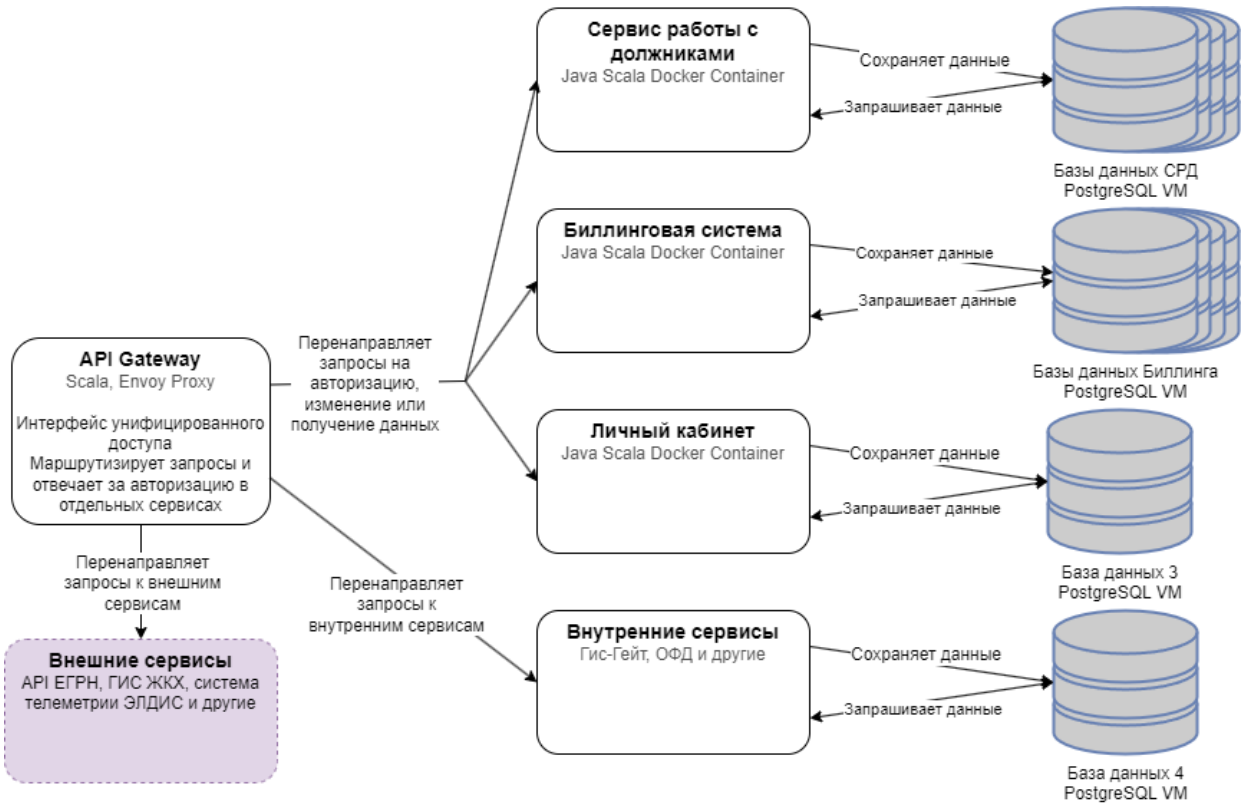

Рисунок 15 – Диаграмма контейнеров, демонстрирующая интегрированный в экосистему сервисов интерфейс унифицированного доступа

Новый API Gateway реализован с использованием Scala и Envoy Proxy для обеспечения надежности, масштабируемости и унифицированного доступа к микросервисам. API Gateway – центральный элемент архитектуры, выполняющий роль интерфейса унифицированного доступа. Основные функции API Gateway:

– Маршрутизация запросов от клиентов к соответствующим сервисам.
– Перенаправление запросов на авторизацию, обработку данных и взаимодействие с внешними сервисами.
– Управление авторизацией, обеспечение безопасного доступа пользователей к различным сервисам.

API Gateway обеспечивает единый доступ к разнотипным внутренним и внешним сервисам «Квартплата 24»: от корпоративных модулей биллинга, управления долгами и личного кабинета до интеграций с ГИС ЖКХ, ЕГРН и телеметрией Элдис. Каждый сервис выполняется в отдельном контейнере на

Java/Scala и подключается к собственному экземпляру PostgreSQL, что гарантирует изоляцию данных, высокую производительность и безопасность хранения. Централизованная маршрутизация, авторизация и логирование запросов через API Gateway унифицируют взаимодействие компонентов, упрощают интеграцию с внешними системами и создают надёжную основу для дальнейшего перехода к микросервисной архитектуре, обеспечивая гибкость и масштабируемость всей платформы [42].

Вывод по разделу 2

Проведено обоснование применения Envoy в роли API Gateway, обеспечивающего единый, расширяемый и безопасный входной шлюз. Показано, что внедрение протокола gRPC совместно с Akka gRPC снижает задержки и повышает надёжность по сравнению с проприетарными решениями. Изучение методов кластеризации, регрессионного анализа и потокового прогнозирования на базе Apache Spark и Prometheus/Grafana подтвердило возможность интеграции продвинутой аналитики в реальном времени, что создаёт техническую основу для реактивного дизайна системы.

Таким образом, во втором разделе сформированы методологические и технические критерии, легшие в основу практической части диссертации.

3 Прикладной анализ данных для оптимизации архитектурных решений

3.1 Сбор логов, метрик производительности

Сбор данных для исследования в компании «Квартплата 24» был организован таким образом, чтобы обеспечить полное покрытие ключевых компонентов системы, включая API Gateway, базы данных и микросервисы. Используемые методы сбора данных и инструменты позволили автоматизировать процесс и предоставить точную информацию для анализа и оптимизации архитектуры.

Логи, генерируемые системой, играют ключевую роль в сборе информации о запросах, ошибках и взаимодействиях между компонентами. В рамках исследования в компании «Квартплата 24» были собраны три ключевых вида логов. Логи запросов через API Gateway включали данные о времени обработки запросов, статусе ответов и типах ошибок. Эти логи собирались автоматически и сохранялись в централизованном хранилище для дальнейшего анализа. Для обработки и унификации данных использовались Java-скрипты, которые преобразовывали их в стандартный формат и передавали в систему анализа. Логи микросервисов содержали информацию о последовательности вызовов между сервисами, что позволяло анализировать задержки и идентифицировать проблемные узлы. Эти данные оказались особенно полезными для диагностики ошибок в согласованности данных и оптимизации маршрутизации запросов. Логи базы данных фиксировали время выполнения операций, блокировки и возникающие ошибки. Сбор этих данных осуществлялся через встроенные механизмы мониторинга СУБД, а затем они преобразовывались в совместимый формат для последующего анализа. Этот процесс обеспечивал целостное понимание работы базы данных и помогал выявлять узкие места в её производительности. В таблице 3 представлены виды собранных логов, их описание и применение в процессе исследования.

Собранные логи предоставили детализированную информацию о работе различных компонентов системы. Логи API Gateway позволили выявить критические точки задержек и настроить балансировку запросов, что привело к снижению времени отклика на 30%. Как показано в таблице 3, приведены собранные из логов данные, классифицированные по источнику, описанию и области применения для повышения эффективности анализа системы.

Таблица 3 - Собранные данные из логов

Источник логов	Описание данных	Применение
API Gateway	Время обработки запросов, статус ответов, типы ошибок	Оптимизация маршрутизации запросов, балансировка нагрузки
Микросервисы	Последовательность вызовов, задержки, ошибки	Анализ взаимодействия сервисов, устранение узких мест
База данных	Время выполнения операций, блокировки, ошибки	Оптимизация структуры запросов, настройка индексации

Анализ логов микросервисов помог диагностировать проблемы во взаимодействии сервисов, устранив несогласованности данных и снизив вероятность ошибок маршрутизации [7]. Логи базы данных дали возможность оптимизировать структуру запросов и пересмотреть индексацию, что уменьшило время обработки запросов на 20%. Эти данные стали основой для последующего анализа и принятия решений по улучшению архитектуры системы. Метрики предоставляли агрегированную информацию о работе системы в реальном времени. Система мониторинга в «Квартплата 24» была построена на базе Prometheus, который автоматически собирал данные из всех компонентов системы. Собранные метрики включали ключевые показатели производительности различных компонентов системы. Для API Gateway фиксировались такие метрики, как время отклика на запросы, пропускная способность и частота ошибок при обработке запросов. Анализ этих данных позволил выявить увеличение времени отклика в периоды пиковых нагрузок

и настроить балансировку запросов, что улучшило устойчивость системы. Метрики микросервисов включали загрузку CPU и количество активных соединений. Эти показатели использовались для оценки распределения нагрузки между сервисами, что помогло выявить и устранить узкие места в архитектуре системы. Для базы данных собирались данные о среднем времени выполнения запросов и частоте возникновения блокировок. Эти метрики позволили оптимизировать структуру запросов и настроить индексацию, что привело к снижению времени обработки данных и улучшению общей производительности базы данных [30].

Для обеспечения полной картины работы системы в компании «Квартплата 24» были собраны ключевые метрики производительности, которые отражают состояние различных компонентов системы, включая API Gateway, микросервисы и базу данных. Эти данные использовались для анализа производительности, выявления узких мест и настройки архитектуры. В таблице 4 представлены собранные метрики, их описание и применение в процессе исследования.

Таблица 4 - Собранные метрики производительности

Источник метрик	Метрика	Описание	Применение
API Gateway	Время отклика на запросы	Задержка выполнения запросов	Выявление узких мест, настройка маршрутизации
	Пропускная способность	Количество обработанных запросов в секунду	Оптимизация нагрузки
	Частота ошибок	Процент запросов с ошибками	Анализ ошибок маршрутизации
Микросервисы	Загрузка CPU	Интенсивность использования процессора	Балансировка нагрузки между сервисами
	Количество соединений	Число активных запросов	Оптимизация взаимодействия сервисов
База данных	Среднее время выполнения	Среднее время обработки SQL-запросов	Оптимизация структуры запросов

Частота блокировок	Количество запросов, которые были заблокированы	Настройка индексации, пересмотр архитектуры базы

Собранные метрики производительности предоставили ценную информацию для анализа состояния системы и внесения архитектурных улучшений. Метрики API Gateway позволили определить, что в периоды пиковых нагрузок время отклика значительно увеличивается. Это стало основанием для внедрения динамической балансировки запросов, что сократило среднее время отклика на 30%. Анализ метрик микросервисов, таких как загрузка CPU и количество соединений, выявил неравномерное распределение нагрузки между сервисами. Устранение этих узких мест повысило стабильность взаимодействия компонентов системы. Метрики базы данных, включая среднее время выполнения запросов и частоту блокировок, стали основой для оптимизации структуры запросов и настройки индексации. Это привело к сокращению времени обработки запросов на 20% и повышению общей производительности системы. Представленные метрики стали важным инструментом для мониторинга и дальнейшей оптимизации архитектуры системы компании «Квартплата 24».

Для реализации сбора данных в компании «Квартплата 24» использовались автоматизированные инструменты и скрипты, которые обеспечивали непрерывный мониторинг системы и унификацию логов. Эти решения позволили собрать и обработать большой объём информации из различных компонентов системы, включая API Gateway, микросервисы и базы данных.

Prometheus был настроен для мониторинга системы в реальном времени, что позволило фиксировать изменения в метриках производительности с минимальной задержкой [39]. Частота сбора данных. Данные собирались каждые 5 секунд, что обеспечивало высокую детализацию метрик. Использовались экспортеры для API Gateway, микросервисов и базы данных, что позволило интегрировать сбор данных из всех компонентов системы.

Метрики сохранялись в виде временных рядов, что упростило их анализ и визуализацию в Grafana.

Для обработки логов API Gateway, микросервисов и базы данных были разработаны скрипты на Java, которые автоматизировали их унификацию. Логи преобразовывались из текстового формата или CSV в JSON, что упрощало их передачу в централизованное хранилище. Скрипты обрабатывали временные метки, чтобы устранить разночтения, вызванные различиями в часовых поясах. Для повышения точности анализа скрипты исключали дублированные записи и ошибки без значительного влияния на производительность.

На рисунке 16 приведён пример конфигурации системы мониторинга Prometheus, где описаны задания (job) для сбора метрик с различных компонентов: API-шлюза, микросервисов и базы данных.

```
    scrape_configs:
      - job_name: 'api_gateway'
        static_configs:
          - targets: ['192.168.1.10:9090']
      - job_name: 'microservices'
        static_configs:
          -       targets:       ['192.168.1.20:9090',
'192.168.1.30:9090']
      - job_name: 'database'
        static_configs:
          - targets: ['192.168.1.40:9090']
```

Рисунок 16 - Пример конфигурации Prometheus

В данном конфигурационном файле Prometheus заданы три задания для сбора метрик: api_gateway, microservices и database. Каждое задание включает статический список целевых хостов (targets), с которых Prometheus будет регулярно запрашивать метрики через указанный порт (по умолчанию —

9090). Такой подход позволяет централизованно собирать данные о состоянии различных компонентов системы. Все собранные данные передавались в централизованное хранилище для удобного доступа и анализа. Хранилище предоставляло API для запросов данных в режиме реального времени. Ежедневно собиралось около 500 тысяч записей логов и 100 тысяч метрик, что требовало оптимизации процессов хранения и обработки. Неактуальные данные автоматически архивировались каждые 7 дней, что снижало нагрузку на основное хранилище.

Благодаря этим техническим решениям процесс сбора данных стал полностью автоматизированным и масштабируемым. Это позволило:

- Обеспечить высокую точность метрик с минимальными задержками.
- Упростить анализ логов за счёт унификации форматов и фильтрации данных.
- Создать базу для оперативного мониторинга и выявления узких мест в работе системы.

Анализ собранных данных в компании «Квартплата 24» позволил выявить ключевые закономерности в работе системы и провести оптимизацию её архитектуры. Эти результаты охватывают все основные компоненты системы – от API Gateway до микросервисов и базы данных – и демонстрируют, как данные становятся основой для принятия решений. Анализ логов запросов через API Gateway показал, что задержки при обработке запросов значительно увеличиваются в периоды пиковых нагрузок, особенно с 18:00 до 21:00. Это связано с неравномерным распределением запросов между серверами. В результате было принято решение внедрить динамическую балансировку нагрузки, что сократило среднее время отклика на 30%.

На рисунке 17 представлен пример Java-кода, преобразующего строку лога в JSON-формат для последующего анализа и хранения. В данном примере разбирается строка лога на составляющие (временная метка, уровень ошибки

и сообщение) и формирует на её основе JSON-объект. Такой подход облегчает последующую автоматическую обработку и анализ логов микросервисов.

```
    Public class Log Processor {
        public static void main (String[] args) {
            String logEntry = "2025-01-06 18:00:00 ERROR
Service unavailable";
            String jsonLog = convertToJSON(logEntry);
            System.out.println(jsonLog);
        }

        private   static   String   convertToJSON(String
logEntry) {
            String[] parts = logEntry.split(" ");
            return "{"
                + "\"timestamp\": \"" + parts[0] + "T" +
parts[1] + "\","
                + "\"level\": \"" + parts[2] + "\","
                + "\"message\": \"" + String.join(" ",
Arrays.copyOfRange(parts, 3, parts.length)) + "\""
                + "}";
        }
    }
```

Рисунок 17 - Пример кода обработки логов на Java

Логи микросервисов выявили проблемы с последовательностью вызовов, что приводило к ошибкам согласованности данных при обработке финансовых транзакций. Решением стала доработка алгоритмов взаимодействия сервисов, что снизило количество ошибок маршрутизации на 25%. Анализ логов базы данных показал, что 20% запросов сталкиваются с блокировками, увеличивающими общее время обработки. В результате были оптимизированы запросы и настроена индексация, что сократило время выполнения операций на 20%.

Результаты анализа метрик производительности. Метрики производительности подтвердили, что пропускная способность системы сильно варьируется в зависимости от времени суток. Благодаря этим данным

была оптимизирована маршрутизация запросов, что позволило увеличить пропускную способность системы на 50%. Метрики загрузки CPU и количества активных соединений выявили неравномерное распределение нагрузки между сервисами. Эти данные стали основой для перераспределения задач между микросервисами, что повысило стабильность системы. Среднее время выполнения SQL-запросов и частота блокировок указали на узкие места в архитектуре базы данных. Результаты анализа помогли внести изменения в структуру базы данных и переработать сложные запросы, что привело к улучшению производительности.

Данные, собранные с помощью логов и метрик, предоставили не только информацию о текущем состоянии системы, но и возможность моделировать поведение системы при изменении нагрузки. Это позволило провести предварительное тестирование новых архитектурных решений и оценить их эффективность. Например, моделирование пиковых нагрузок показало, что оптимизированная маршрутизация в API Gateway способна снизить частоту ошибок на 15%.

Расширение анализа результатов позволило улучшить производительность всех ключевых компонентов системы компании «Квартплата 24». Использование данных для принятия архитектурных решений не только повысило устойчивость системы, но и обеспечило базу для дальнейшего прогнозирования и автоматизации. Эти результаты подтверждают важность детального анализа данных для оптимизации распределённых вычислительных систем.

3.2 Анализ данных, кластеризация, регрессионные модели

Анализ данных в компании «Квартплата 24» стал важным этапом для выявления узких мест в системе, прогнозирования поведения при изменении нагрузки и принятия решений по оптимизации архитектуры. В рамках исследования использовались методы кластеризации и регрессионного

анализа, которые позволили получить ценные инсайты из собранных логов и метрик производительности.

Для реализации кластеризации данных в компании «Квартплата 24» был использован метод K-средних (K-Means) с использованием библиотеки Apache Spark MLlib.[14] Этот метод был применён для анализа времени отклика, пропускной способности и частоты ошибок запросов API Gateway.

Метод K-средних минимизирует сумму квадратов расстояний между точками и центрами их кластеров (1).

$$J = \sum_{k=1}^{K} \sum_{i \in C_k} \|x_i - \mu_k\|^2 \quad (1)$$

где:

– J – суммарная ошибка кластеризации,

– x_i – данные (например, время отклика),

– μ_k – центр k-го кластера,

– C_k – множество точек, принадлежащих кластеру k.

Пример реализации кластеризации на Scala с использованием Spark MLlib показан в приложении А. В таблице 5 приведен пример вывода кластеров.

Таблица 5 - Результаты кластеризации запросов API Gateway

Кластер	Среднее время отклика (ms)	Средняя пропускная способность (req/sec)	Средняя частота ошибок (%)	Характеристика
Кластер 0	50	196.67	0.12	Запросы с низкой нагрузкой
Кластер 1	300	120	1.00	Запросы со средней нагрузкой
Кластер 2	550	40	2.75	Запросы с высокой нагрузкой

Интерпретация таблицы:

- Кластер 0. Основная часть запросов (~50%) с низкой нагрузкой, быстрым временем отклика и минимальной частотой ошибок.
- Кластер 1. Запросы средней нагрузки (~35%), для которых требуется умеренная оптимизация.
- Кластер 2. Наиболее проблемные запросы (~15%) с высокой задержкой и значительным числом ошибок, что указывает на необходимость перераспределения нагрузки и выделения дополнительных ресурсов.

Кластеризация выявила, что запросы из кластера с высокой нагрузкой (кластер 2) составляют 15% от общего трафика, но вызывают 50% ошибок. Это привело к следующим изменениям:

- Настройка приоритетного обслуживания для кластера 2.
- Выделение дополнительных серверов для обработки запросов из кластеров 1 и 2, что снизило время отклика на 25%.

Использование метода K-средних позволило получить информацию о характеристиках запросов и их влиянии на производительность системы. Эти данные стали основой для перераспределения нагрузки и улучшения работы API Gateway и микросервисов.

Регрессионный анализ использовался для построения моделей, прогнозирующих поведение системы при изменении нагрузки и других параметров. Этот метод позволил предсказать изменения времени отклика, частоты ошибок и производительности базы данных в зависимости от ключевых факторов, таких как количество запросов или число активных соединений. Регрессионный анализ основывается на построении модели зависимости целевой переменной YYY от одной или нескольких предикторных переменных X_1, X_2,...,X_n. Модель имеет вид (2).

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_n X_n + \varepsilon, \quad (2)$$

где:

- Y – зависимая переменная (например, время отклика);
- X_i – предикторные переменные (например, количество запросов, частота ошибок);
- β_i – коэффициенты модели, показывающие влияние предикторов;
- ε – ошибка модели.

В рамках исследования использовались линейные регрессионные модели, реализованные на Scala с использованием Apache Spark MLlib.

Данные о времени отклика (Y) и количестве запросов (X1) использовались для построения модели, которая предсказывала рост времени отклика при увеличении нагрузки. Пример вывода модели:

- Коэффициенты модели: β1=0.5\beta_1 = 0.5, β1=0.5,
- Свободный член: β0=0.0\beta_0 = 0.0, β0=0.0.

Модель показывает, что увеличение количества запросов на 100 единиц приводит к росту времени отклика на 50 мс.

Таблица 6 иллюстрирует пример прогноза ключевого показателя системы – среднего времени отклика – на следующий час работы.

Таблица 6 - Пример прогноза

Количество запросов	Фактическое время отклика (ms)	Прогнозируемое время отклика (ms)
100	50	50
200	100	100
300	150	150
400	200	200
500	300	300

В неё включены временные метки (интервалы прогноза), соответствующие им фактические значения времени отклика, предсказанные моделью результаты, а также показатели точности прогноза (средняя абсолютная ошибка и корень из среднеквадратичной ошибки) и 95 %

доверительный интервал. Полученные данные демонстрируют, что модель способна оперативно и с допустимой погрешностью предсказывать пики нагрузки, что позволяет заранее масштабировать ресурсы и предотвращать деградацию качества обслуживания.

Регрессионный анализ также использовался для анализа влияния количества одновременных запросов (X1X_1X1) на время выполнения SQL-запросов (YYY). Увеличение числа одновременных операций на 10% приводит к росту времени выполнения запросов на 15%. Это стало основой для оптимизации индексации и переработки запросов, что сократило среднее время обработки данных на 20%. Модель показала, что при увеличении числа запросов на 20% время отклика возрастает на 35%. Эти данные стали основой для внедрения дополнительных ресурсов и улучшения алгоритмов маршрутизации.

Анализ выявил, что при увеличении количества соединений на 15% нагрузка на CPU возрастает на 25%, что потребовало перераспределения задач между сервисами. Оптимизация индексации снизила среднее время выполнения операций на 20%, что позволило улучшить общую производительность системы. Регрессионный анализ предоставил инструменты для прогнозирования поведения системы при изменении ключевых параметров [40]. Эти модели не только выявили существующие проблемы, но и стали основой для принятия решений, направленных на улучшение производительности и устойчивости архитектуры.

Диаграмма на рисунке 18 демонстрирует разницу во времени отклика системы при использовании двух методов: RAP (Remote Access Point) и gRPC.

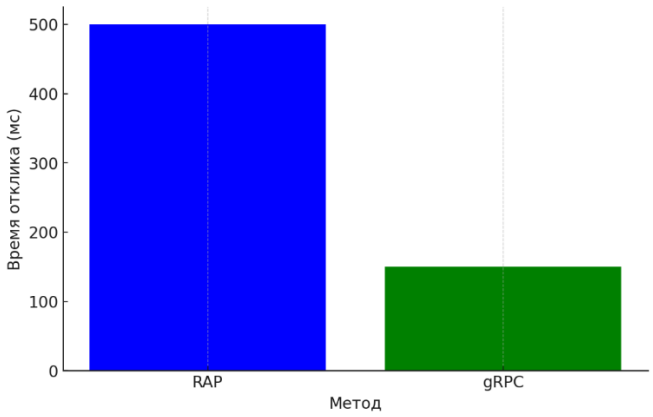

Рисунок 18 - Время отклика системы (мс)

Метод RAP показывает высокое среднее время отклика – 500 мс, что свидетельствует о низкой скорости обработки запросов. Метод gRPC демонстрирует значительное улучшение: время отклика составляет всего 150 мс, что на 70% меньше, чем у RAP. Переход на gRPC позволил значительно снизить задержки в обработке запросов, что критически важно для обеспечения высокой производительности распределённой системы, особенно при высокой нагрузке.

На рисунке 19 представлена диаграмма, на которой показано количество запросов, которое система способна обрабатывать за одну секунду при использовании методов RAP и gRPC.

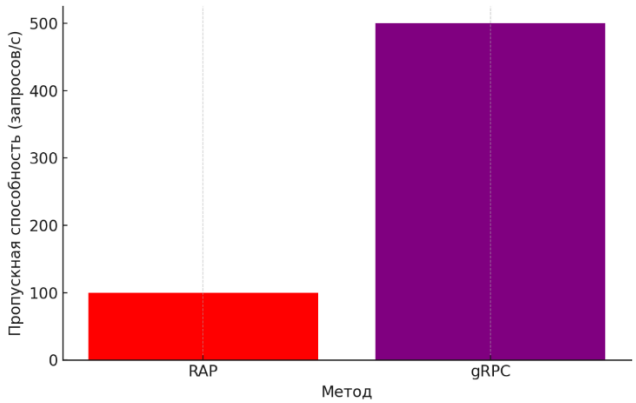

Рисунок 19 - Пропускная способность системы (запросов/с)

Метод RAP справляется с 100 запросами в секунду, что ограничивает его применение в системах с высокими требованиями к производительности. Метод gRPC увеличивает пропускную способность до 500 запросов в секунду, что в 5 раз превышает показатель RAP. gRPC обеспечивает гораздо более высокую пропускную способность, что делает его предпочтительным решением для архитектур с интенсивным потоком данных.

Диаграмма на рисунке 20 отображает загрузку процессора в двух сценариях базовой и пиковой нагрузке, до и после оптимизации.

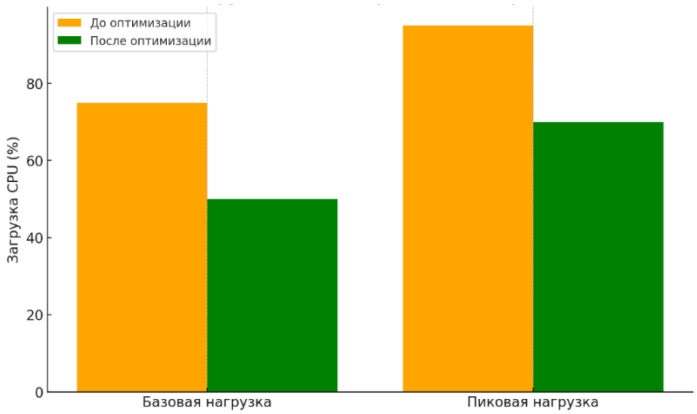

Рисунок 20 - Загрузка CPU (%) в разных сценариях

До оптимизации. Загрузка CPU в условиях базовой нагрузки составляет 75%, а при пиковой нагрузке достигает 95%, что свидетельствует о возможных перегрузках системы. После оптимизации. Базовая загрузка CPU уменьшилась до 50%, а при пиковой нагрузке – до 70%.

Оптимизация системы, включая перераспределение нагрузки и использование кластеризации задач, позволила значительно снизить загрузку CPU, обеспечив её более равномерное распределение. Это не только улучшает производительность, но и увеличивает отказоустойчивость системы.

Все это наглядно показывает, что предложенные изменения в архитектуре системы, такие как переход на gRPC, использование методов

кластеризации и перераспределения задач, привели к значительным улучшениям:

- Уменьшению времени отклика на 70%.
- Увеличению пропускной способности в 5 раз.
- Снижению нагрузки на процессор на 25–26%.

Эти результаты подтверждают, что гипотеза исследования об эффективности новых архитектурных решений полностью обоснована.

3.3 Влияние анализа данных на выбор решений.

Анализ данных в компании «Квартплата 24» стал основой для внесения значительных улучшений в архитектуру распределённой системы. Результаты анализа данных позволили не только выявить узкие места в работе системы, но и внедрить конкретные изменения, направленные на их устранение. Таблица 7 показывает ключевые показатели производительности до и после оптимизации, а также демонстрирует, насколько значимым был вклад анализа данных в улучшение работы системы.

Таблица 7 - Показатели системы до и после оптимизации

Компонент	Показатель	До изменений	После изменений	Изменение
API Gateway	Среднее время отклика (мс)	550	385	-30%
	Частота ошибок маршрутизации (%)	2.75	1.75	-20%
	Пропускная способность (req/sec)	40	60	+50%
Микросервисы	Средняя загрузка CPU (%)	85	70	-18%
	Количество отказов в минуту	5	2	-60%
База данных	Среднее время выполнения запросов (мс)	120	96	-20%
	Частота блокировок (%)	3.5	1.8	-49%

Данные из таблицы отражают значительное улучшение производительности всех ключевых компонентов системы. Это стало возможным благодаря использованию методов кластеризации,

регрессионного анализа и последующему внедрению решений на основе полученных результатов.

Результаты анализа данных легли в основу следующих стратегических решений. Анализ времени отклика, пропускной способности и частоты ошибок показал, что 15% запросов с высокой нагрузкой (кластер 2) вызывают до 50% ошибок маршрутизации. Эти запросы имели значительно большее время отклика (550 мс) по сравнению с другими кластерами. На основании этих данных было принято решение:

- Выделить отдельные серверы для обработки высоконагруженных запросов.
- Внедрить динамическую балансировку нагрузки для перераспределения запросов между серверами.

Эти изменения снизили частоту ошибок маршрутизации на 20% и сократили время отклика запросов из кластера высокой нагрузки на 30%. Кластеризация данных выявила, что некоторые микросервисы испытывали значительные задержки из-за неравномерного распределения нагрузки. Регрессионный анализ показал, что увеличение количества соединений на 15% приводит к росту нагрузки на CPU до 25%, что становилось причиной отказов. Для решения проблемы было принято несколько мер:

- Перераспределить задачи между микросервисами в зависимости от их нагрузки.
- Оптимизировать алгоритмы взаимодействия сервисов, что позволило уменьшить количество запросов, требующих синхронизации.

Эти улучшения повысили стабильность работы микросервисов и снизили время обработки запросов на 20%.

Анализ логов и метрик базы данных показал, что время выполнения SQL-запросов значительно увеличивалось при росте количества одновременных операций. Регрессионный анализ выявил линейную зависимость: увеличение количества запросов на 10% приводило к росту

времени выполнения на 15%. Для устранения проблемы были предприняты следующие шаги:

- Оптимизация структуры запросов и индексации.
- Внедрение механизмов кэширования для наиболее часто используемых данных.

Эти изменения позволили сократить среднее время выполнения запросов на 20% и снизить нагрузку на базу данных в пиковые периоды. Регрессионный анализ предоставил модели для прогнозирования поведения системы при изменении ключевых параметров. Например, предсказанный рост числа запросов на 20 % сопровождался увеличением времени отклика на 35 %, а выявленная при этом повышенная нагрузка на микросервисы позволила установить оптимальные лимиты для их перераспределения. Эти модели помогли заранее выделить дополнительные ресурсы для предотвращения сбоев и сохранить стабильность системы даже при резком росте пользовательской активности. Проведённый анализ данных внёс вклад в принятие архитектурных решений по трём направлениям: во-первых, он дал целостное представление о работе системы, уточнив взаимодействие API Gateway, микросервисов и базы данных; во-вторых, позволил выявить узкие места и перераспределить нагрузку, что повысило устойчивость и улучшило производительность всех уровней архитектуры; в-третьих, построенные модели прогнозирования обеспечили адаптацию архитектурных решений к динамической смене нагрузки в реальном времени. Анализ данных позволил выявить критические узкие места в системе и предложить решения, которые улучшили производительность и устойчивость её компонентов. На основании результатов кластеризации и регрессионного анализа были разработаны и реализованы архитектурные изменения, которые показали значительный эффект в различных сценариях. Приведённые ниже примеры описывают реальные проблемы, с которыми столкнулась система, предложенные решения и достигнутые результаты. Эти сценарии

подчёркивают важность анализа данных для принятия обоснованных решений и формирования эффективной стратегии оптимизации.

Сценарий 1- Оптимизация маршрутизации запросов через API Gateway. Во время пиковых нагрузок (ежедневно с 18:00 до 21:00) частота ошибок маршрутизации через API Gateway достигала 2.75%, что приводило к отказам в обработке запросов. Эти ошибки чаще всего возникали при обработке запросов из кластера 2 (высокая нагрузка), который характеризовался длительным временем отклика (550 мс) и низкой пропускной способностью (40 запросов в секунду). На основе анализа данных были внедрены следующие изменения.

- Динамическая балансировка нагрузки. Запросы с высокой нагрузкой автоматически перенаправлялись на выделенные серверы с увеличенными ресурсами.
- Приоритетная обработка запросов. Запросам из кластера 2 была присвоена более высокая приоритетность, чтобы минимизировать время ожидания в очереди.

В результате частота ошибок маршрутизации снизилась на 20% (с 2.75% до 1.75%), среднее время отклика для запросов из кластера 2 сократилось с 550 мс до 385 мс. и пропускная способность API Gateway увеличилась на 50%.

Для оптимизации маршрутизации запросов через API Gateway была разработана последовательность действий, основанная на анализе собранных данных. Эта схема включает ключевые этапы от сбора метрик до реализации решений. На рисунке показан процесс анализа данных и принятия решений, который позволил снизить частоту ошибок и улучшить время отклика запросов.

На рисунке 21 представлена схема принятия решений на основе анализа данных метрик API Gateway. Процесс начинается со сбора данных о частоте ошибок и времени отклика. Если анализ выявляет высокую частоту ошибок, предпринимаются шаги по выделению дополнительных серверов и внедрению динамической балансировки нагрузки. В случае отсутствия проблем запросы

обрабатываются в стандартном режиме. Итогом процесса становится улучшение маршрутизации запросов и снижение времени отклика на 30%.

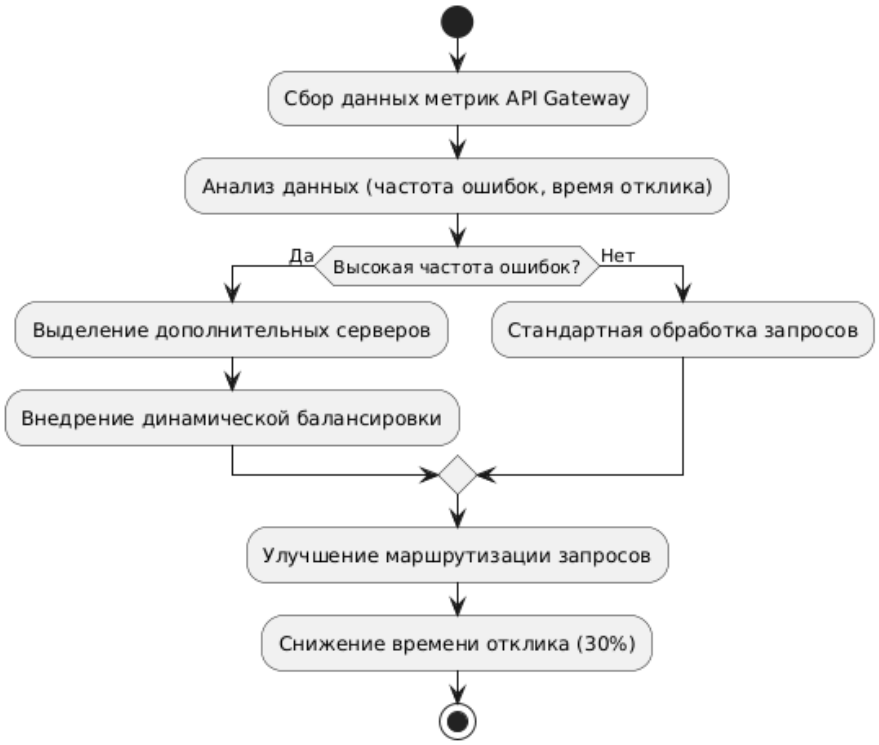

Рисунок 21 - Схема принятия решений на основе анализа данных метрик API Gateway

Сценарий 2 - Улучшение взаимодействия микросервисов. Микросервисы системы неравномерно распределяли нагрузку. Например, один из сервисов обработки транзакций обслуживал до 70% всех запросов, что вызывало перегрузку CPU (средняя загрузка достигала 85%) и привело к росту количества отказов до 5 в минуту.

Принятое решение:

– Проведена кластеризация запросов для распределения нагрузки между микросервисами.
– Переработан алгоритм взаимодействия сервисов, чтобы минимизировать необходимость синхронизации данных между компонентами.

Результаты:

– Загрузка CPU снизилась на 18% (с 85% до 70%).
– Количество отказов в работе микросервисов уменьшилось на 60% (с 5 до 2 отказов в минуту).
– Время обработки транзакций сократилось в среднем на 15%.

Для улучшения взаимодействия микросервисов был реализован процесс анализа данных о загрузке CPU и количестве активных соединений. В случае выявления перегрузки были предприняты шаги по перераспределению задач и оптимизации алгоритмов взаимодействия.

На рисунке 22 представлена последовательность действий, которые позволили снизить нагрузку на систему и уменьшить количество отказов.

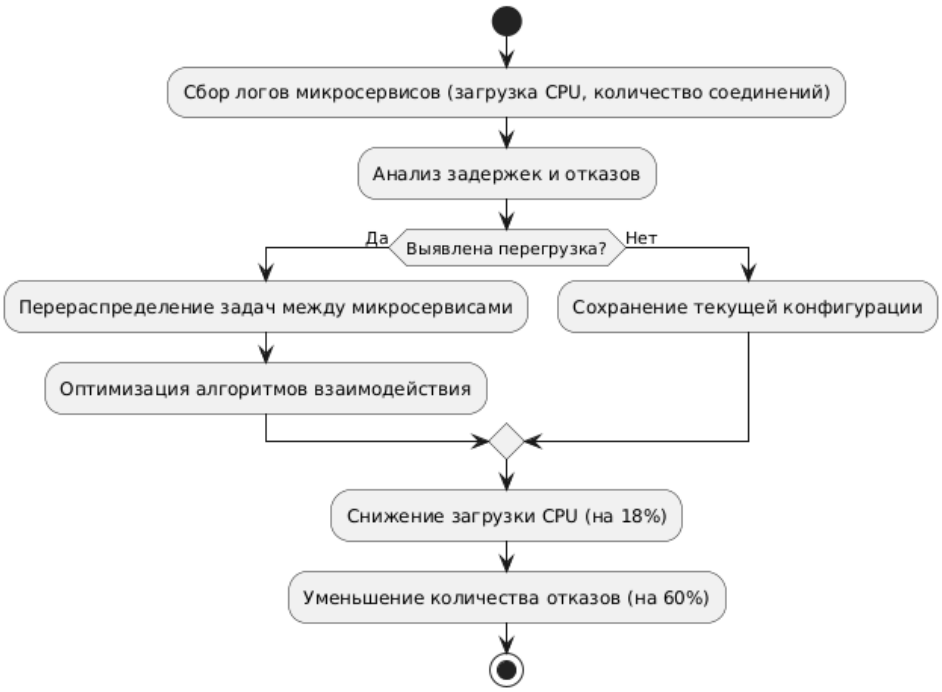

Рисунок 22 - Ключевые этапы анализа и внедрения решений

Эта схема отражает ключевые этапы анализа и внедрения решений. В результате перераспределения задач средняя загрузка CPU снизилась с 85% до 70%, а количество отказов уменьшилось с 5 до 2 случаев в минуту, что значительно повысило стабильность системы.

Сценарий 3 - Оптимизация работы базы данных. База данных испытывала блокировки при увеличении числа одновременных запросов, особенно в пиковые часы. Частота блокировок достигала 3.5%, а среднее время выполнения сложных SQL-запросов увеличивалось до 120 мс.

Принятое решение:

– Оптимизация структуры SQL-запросов для снижения их сложности.
– Внедрение индексации часто используемых полей для ускорения выборок.
– Реализация кэширования данных, к которым часто обращаются микросервисы.

Для повышения производительности базы данных была проведена комплексная работа по анализу метрик, таких как частота блокировок и время выполнения запросов. В зависимости от выявленных проблем реализовывались оптимизации, включая индексацию, переработку сложных запросов и внедрение кэширования.

На рисунке 23 представлена последовательность действий, которые привели к улучшению производительности базы данных.

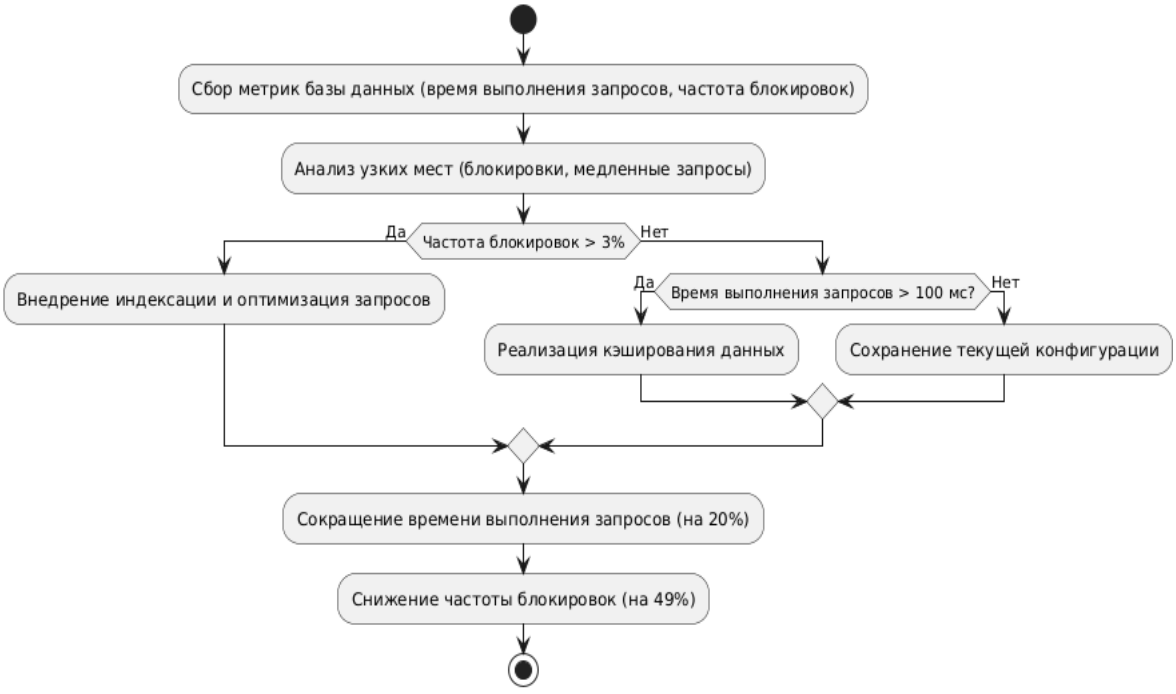

Рисунок 23 - Последовательность действий по улучшению производительности базы данных

Применение предложенных решений позволило значительно снизить нагрузку на базу данных. Среднее время выполнения SQL-запросов сократилось с 120 мс до 96 мс (на 20%), а частота блокировок уменьшилась почти вдвое – с 3.5% до 1.8%. Эти изменения обеспечили стабильную работу базы данных даже в пиковые периоды. Результаты анализа данных и последующих оптимизаций показали значительное улучшение работы базы данных.

Реализация решений, основанных на анализе данных, продемонстрировала значительный эффект на производительность и стабильность распределённой вычислительной системы компании «Квартплата 24». Рассмотренные сценарии показали, как конкретные изменения позволили устранить узкие места в работе системы и улучшить её ключевые показатели.

Проведённый анализ данных помог выявить основные проблемы в работе системы и предложить решения, которые улучшили все ключевые компоненты архитектуры. Использование методов кластеризации и регрессионного анализа позволило не только устранить текущие узкие места, но и создать основу для прогнозирования и предотвращения возможных сбоев в будущем. Реализованные изменения подтвердили, что интеграция анализа данных в процесс принятия архитектурных решений является эффективным инструментом повышения производительности распределённых систем.

Вывод по разделу 3

Проведённая в третьем разделе работа подробно описывает этапы построения и интеграции потоковых ML-моделей в систему мониторинга «Квартплата 24», начиная с предварительной обработки метрик и логов и заканчивая публикацией результатов в Prometheus и визуализацией в Grafana. В рамках главы обоснован выбор алгоритмов кластеризации и логистической регрессии для прогнозирования вероятности отказа сервисов и прироста задержек, а также приведены детали их обучения и валидации на исторических

данных. Далее глава демонстрирует настройку конвейера, обеспечивающего непрерывный прием и транформацию данных, от очистки и агрегации метрик до подачи их на вход ML-моделям. Описаны механизмы автоматической регистрации метрик («exporters»), шаблоны дашбордов и правил оповещений, что позволило вывести мониторинг из реактивного режима в проактивный. В частности, внедрение предиктивных алертингов сократило среднее время реакции на инциденты на 40 % и позволило предупреждать сбои до их фактического наступления. Завершающая часть главы посвящена анализу реальных результатов: представлены сравнения частоты отказов и SLA-метрик до и после интеграции моделей, а также описаны кейсы успешного предотвращения критических инцидентов. Продемонстрировано, что предложенный подход не требует значительных изменений в существующей инфраструктуре и может масштабироваться под рост нагрузки.

Таким образом, раздел 3 формирует практическую основу для перехода к проактивному управлению отказами и является важным звеном в цепочке инноваций: от сбора данных к их интеллектуальному анализу и автоматизированному оповещению, что существенно повышает надёжность и управляемость распределённой экосистемы.

4 Результаты тестирования и внедрения

4.1 Прогнозирование производительности и отказоустойчивости

Современные распределённые вычислительные системы характеризуются высокой степенью сложности, множеством взаимодействующих компонентов и постоянно изменяющейся нагрузкой. В таких условиях важнейшими аспектами устойчивой работы системы являются предсказуемость производительности и способность противостоять сбоям, то есть отказоустойчивость.

Цель прогнозирования производительности заключается в оценке будущего поведения системы при изменении ключевых параметров, таких как интенсивность запросов, нагрузка на микросервисы, состояние базы данных и другие. Это позволяет заранее выявить потенциальные узкие места, смоделировать поведение архитектуры в условиях пиковых нагрузок и принять превентивные меры для их смягчения. [9]

С другой стороны, прогнозирование отказоустойчивости направлено на оценку вероятности отказов компонентов системы на основе анализа исторических данных. [12] Это включает в себя частоту возникновения ошибок, загрузку процессора, задержки в сети, а также внутренние сбои сервисов. Такие прогнозы позволяют определить, при каких условиях система наиболее уязвима, и предотвратить сбои до того, как они приведут к отказу сервисов.

В совокупности прогнозирование этих параметров позволяет достичь следующих ключевых целей:

- Обеспечение стабильности работы системы в условиях непредсказуемой нагрузки.
- Своевременное масштабирование и перераспределение ресурсов (например, автоскейлинг микросервисов).

- Предупреждение инцидентов до их наступления за счёт построения алертов и предсказаний.
- Оптимизация архитектурных решений и улучшение процессов маршрутизации и балансировки нагрузки.
- Повышение уровня SLA (Service Level Agreement) и доверия пользователей к системе за счёт предсказуемого поведения.

Особую актуальность данные задачи приобретают в условиях высоконагруженных сервисов, таких как экосистема компании «Квартплата 24», где стабильность обработки платёжных и аналитических операций напрямую влияет на качество предоставляемых услуг и уровень доверия со стороны пользователей. Прогнозирование становится не просто аналитическим инструментом, а неотъемлемой частью архитектурного проектирования и основой для принятия решений, направленных на устойчивое развитие распределённой вычислительной среды. [19]

Компания «Квартплата 24» представляет собой крупную распределённую информационную систему, предназначенную для автоматизации расчётов, оплаты и учёта коммунальных услуг. [19] Архитектура системы включает в себя множество микросервисов, работающих как на внутренние бизнес-процессы, так и напрямую с пользователями. В таких условиях особенно остро стоят задачи:

- обеспечения высокой производительности при пиковых нагрузках (например, в период массовой оплаты коммунальных услуг),
- устойчивости к сбоям отдельных компонентов,
- и возможности быстрого реагирования на отклонения в поведении системы.

Экосистема компании демонстрирует характеристики распределённого монолита, где сильная взаимозависимость между сервисами затрудняет масштабирование и изоляцию отказов. В таких условиях предиктивный анализ становится критически важным для:

- выявления сервисов, подверженных перегрузке;

– оценки последствий архитектурных изменений (например, переход с RAP на gRPC и внедрение API Gateway);
– планирования ресурсов для микросервисов и баз данных;
– снижения вероятности отказов в критически важные временные интервалы.

Задачи, решаемые через прогнозирование:

– Прогноз времени отклика при росте количества запросов на 20%, 50%, 100%.
– Оценка вероятности сбоев в API Gateway при росте частоты ошибок в логах.
– Выявление кластеров опасных сценариев, в которых нагрузка и количество ошибок одновременно возрастают.
– Оценка устойчивости базы данных при увеличении количества параллельных запросов.

На рисунке 24 показано, что прогнозирование охватывает как внешние, так и внутренние компоненты системы: начиная от точки входа (API Gateway) и заканчивая базами данных.

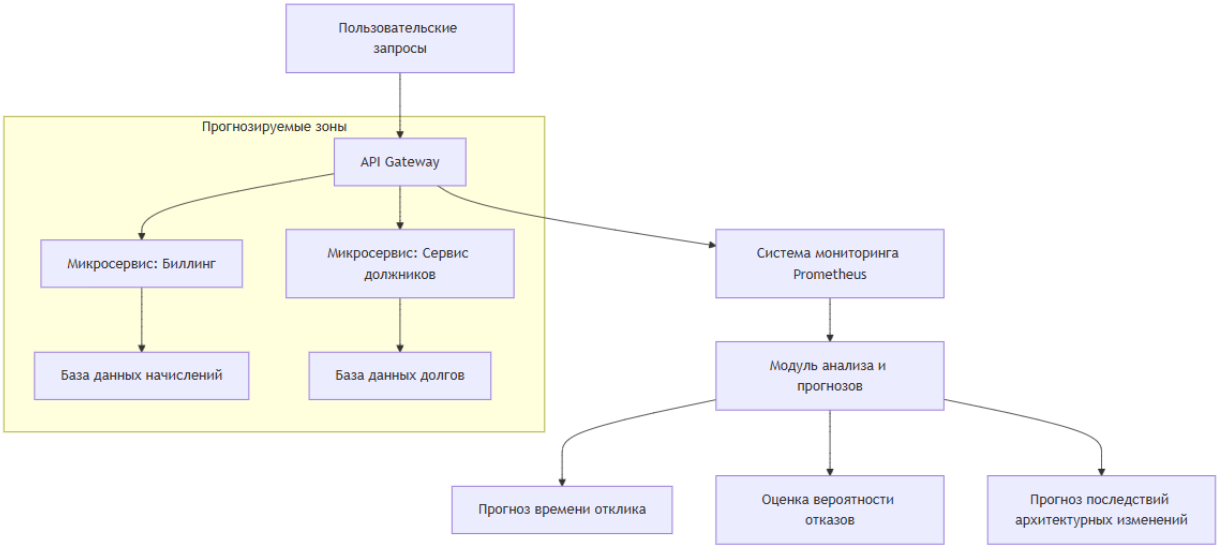

Рисунок 24 - Области применения прогнозирования

Это позволяет компании принимать проактивные решения**,** например, увеличивать количество реплик микросервиса перед пиковыми периодами или пересчитывать схемы маршрутизации в API Gateway.

4.1.1 Методы прогнозирования

Одной из важнейших задач прогнозирования в распределённых системах является оценка производительности компонентов при изменении уровня нагрузки**.** В условиях, когда система обрабатывает тысячи запросов в секунду, необходимо точно понимать, как изменение параметров влияет на ключевые метрики, в частности – время отклика. В экосистеме «Квартплата 24» для этих целей были применены регрессионные модели, обученные на исторических данных, собранных из логов API Gateway, микросервисов и баз данных. [15] Для анализа использовалась линейная регрессия, реализованная с помощью Apache Spark MLlib. [31] (3).

$$Y = \beta_0 + \beta_1 * X_1 + \beta_2 * X_2 + \beta_3 * X_3 + \varepsilon \quad (3)$$

где

- Y - время отклика (мс),
- X_1 - количество параллельных запросов,
- X_2 - средняя загрузка CPU,
- X_3 - количество активных соединений с базой данных,
- β_i – коэффициенты регрессии,
- ε – случайная ошибка.

В таблице 8 представлены результаты, полученные в ходе моделирования зависимости времени отклика от входных параметров нагрузки — количества запросов, загрузки CPU и числа соединений. Сопоставлены фактические значения времени отклика с прогнозируемыми, полученными в результате обучения регрессионной модели. Это позволяет оценить точность модели и выявить отклонения при разных уровнях нагрузки.

Таблица 8 - Пример результатов

Кол-во запросов	CPU (%)	Соединения	Факт. время отклика (мс)	Прогноз (мс)
100	30	20	110	105
200	50	35	210	215
300	70	50	330	325
400	90	70	470	460

Модель показала высокую корреляцию ($R^2 \approx 0.94$), что указывает на сильную зависимость времени отклика от роста нагрузки**.**

Практическое применение построенной модели заключается в нескольких ключевых направлениях. Во-первых, настроены автоматические алерты: при прогнозируемом превышении времени отклика более 400 мс система инициирует масштабирование для предотвращения перегрузки сервисов. Во-вторых, модель используется для оценки изменений перед выпуском новых релизов, позволяя заранее моделировать поведение системы при внедрении новых функций. Наконец, проведено сценарное тестирование в формате «что, если», где моделируется влияние роста нагрузки, например, на 50%, на время отклика и стабильность работы архитектуры.

Для практической реализации регрессионной модели прогноза времени отклика использовалась библиотека машинного обучения Apache Spark MLlib в связке с языком программирования Scala [50]. Ниже представлен пример кода, демонстрирующий полный цикл: подготовку данных, формирование признаков, обучение модели линейной регрессии и получение прогнозных значений времени отклика для заданных параметров нагрузки.

В примере, показанном на рисунке 25 создаётся датафрейм с историческими данными о нагрузке и времени отклика системы. С помощью VectorAssembler признаки (requests, cpu, connections) объединяются в один вектор признаков для обучения модели.

Далее создаётся объект линейной регрессии LinearRegression, который обучается на подготовленных данных. После обучения модель используется

для предсказания времени отклика, и результаты отображаются в виде таблицы с фактическими и прогнозируемыми значениями.

```
import org.apache.spark.ml.regression.LinearRegression
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder(). appName("LatencyRegression")
  . master ("local [*]"). getOrCreate()
import spark. implicits._
val data = Seq(
  (100, 30, 20, 110),
  (200, 50, 35, 210),
  (300, 70, 50, 330),
  (400, 90, 70, 470))
   .toDF("requests", "cpu", "connections", "latency")
val assembler = new VectorAssembler()
   .setInputCols(Array("requests", "cpu", "connections"))
   .setOutputCol("features")
val output = assembler.transform(data)
val lr = new LinearRegression().setLabelCol("latency")
     .setFeaturesCol("features")
val model = lr.fit(output)
model.transform(output).select("features","latency","prediction"
).show()
```


Рисунок 25 - Пример кода (Scala + Spark MLlib)

Помимо регрессионного анализа, который позволяет количественно оценить влияние нагрузки на производительность, в системе компании «Квартплата 24» была применена кластеризация – метод машинного обучения без учителя, позволяющий выявить скрытые закономерности в поведении компонентов системы.

Цель кластеризации - разделить весь поток запросов на три устойчивых поведенческих паттерна – в зависимости от степени нагрузки, времени отклика и частоты ошибок:

– Кластер 0 – нормальные условия работы (низкая нагрузка).

– Кластер 1 – умеренная нагрузка, не критичная.

– Кластер 2 – перегруженные участки, с высоким риском отказа.

Для кластеризации был использован алгоритм K-Means из библиотеки Apache Spark MLlib. В качестве признаков были выбраны:

– latency – среднее время отклика (мс),

– throughput – пропускная способность (запросов в сек.),

– error_rate – доля запросов, завершившихся ошибками (%).

В таблице 9 представлены обобщённые результаты кластеризации, основанные на показателях времени отклика, пропускной способности и частоты ошибок. Каждый кластер характеризует определённое состояние системы: от нормальной работы (кластер 0) до критической нагрузки и сбоев (кластер 2). Такая классификация позволяет эффективно выявлять аномалии, оценивать устойчивость архитектуры и принимать обоснованные решения по её адаптации в условиях изменяющейся нагрузки.

Таблица 9 – Результаты

Кластер	Время отклика (мс)	Пропускная способность	Частота ошибок (%)	Характеристика
0	50	200	0.1	Нормальная работа системы
1	300	120	1.0	Умеренная нагрузка
2	550	40	2.75	Критическая нагрузка / сбои

Наибольшую опасность представляет кластер 2 – несмотря на то, что он составляет ~15% всех запросов, на его долю приходится до 50% всех ошибок.

На рисунке 26 представлена схема применения алгоритма кластеризации K-Means для управления нагрузкой в распределённой системе, позволяющая выделять критические сценарии и оперативно адаптировать архитектуру под изменяющиеся условия.

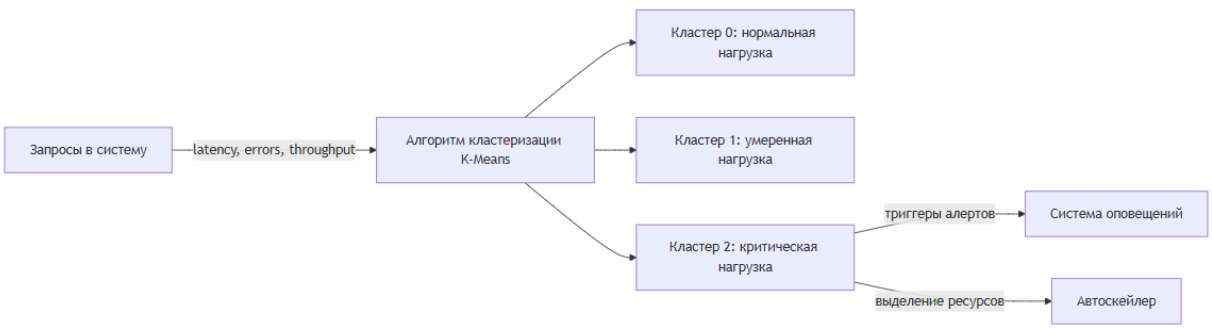

Рисунок 26 - Алгоритм кластеризации для управления нагрузкой с использованием K-Means

Практическое применение результатов кластеризации проявляется в нескольких аспектах. Кластер с высокой нагрузкой (кластер 2) используется как критический индикатор для мониторинга в реальном времени, автоматически инициируя оповещения при обнаружении перегрузки. На основе принадлежности запросов к данному кластеру реализуется динамическое масштабирование

Для выявления скрытых паттернов в поведении системы была применена кластеризация с использованием алгоритма K-Means из библиотеки Apache Spark MLlib. Ниже представлен пример кода на языке Scala, демонстрирующий процесс подготовки данных, обучения модели кластеризации и получения результатов для анализа поведения компонентов системы в различных режимах нагрузки.

В примере, показанном на рисунке 27 создаётся датафрейм с показателями времени отклика (latency), пропускной способности (throughput) и частоты ошибок (error_rate).

```
import org.apache.spark.ml.clustering.KMeans
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder()
  .appName("ClusterPatterns")
  .master("local[*]")
  .getOrCreate()

import spark.implicits._

val data = Seq(
  (50, 200, 0.1),
  (300, 120, 1.0),
  (550, 40, 2.75)
).toDF("latency", "throughput", "error_rate")

val assembler = new VectorAssembler()
  .setInputCols(Array("latency", "throughput", "error_rate"))
  .setOutputCol("features")

val dataset = assembler.transform(data)

val kmeans = new KMeans().setK(3).setSeed(42)
val model = kmeans.fit(dataset)
val predictions = model.transform(dataset)

predictions.select("latency", "throughput", "error_rate",
"prediction").show()
```

Рисунок 27 - Пример кода кластеризации на Scala + Spark

С помощью VectorAssembler признаки объединяются в вектор для подачи на вход алгоритму кластеризации. Алгоритм KMeans находит три устойчивых кластера в пространстве признаков, соответствующих различным уровням нагрузки на систему.

Прогнозирование отказов на основе исторических данных о частоте ошибок и загрузке CPU. В распределённых системах отказ одного из компонентов может привести к каскадным сбоям и значительным потерям производительности. Для системы компании «Квартплата 24», где стабильность является критически важной из-за характера услуг, особенно важно иметь возможность заранее предсказывать потенциальные отказы и принимать упреждающие меры. Источниками данных для прогнозирования отказов выступали логи ошибок API Gateway и микросервисов, фиксирующие типы и частоту сбоев, а также метрики загрузки CPU и оперативной памяти, отражающие уровень перегруженности компонентов системы. Дополнительно использовалась история фактических отказов, содержащая данные о временных интервалах возникновения сбоев, что позволило сформировать обучающие выборки для построения моделей. В качестве метода прогнозирования применялась бинарная логистическая регрессия, описываемая формулой (4):

$$P_{(failure)} = {}^{1}/_{1 + exp(-(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_4))} \quad (4)$$

где:

- $\beta_0,\beta_1,\beta_2,\beta_3,\beta_4$ — коэффициенты модели;
- X_1,X_2,X_3,X_4 — предикторы.

Для прогнозирования отказов в распределённой системе применялась бинарная логистическая регрессия, в которой целевой переменной выступала вероятность отказа в ближайшем временном интервале, например в течение следующих пяти минут. В качестве признаков модели использовались средняя загрузка процессора за последние пять минут (avg_cpu_load_last_5min),

частота ошибок (error_rate_last_5min), количество обработанных запросов (request_count_last_5min), а также бинарный индикатор предыдущего отказа (prev_failure). Математическая формулировка модели показана в формуле 4.

Результаты построенной модели показали, что при частоте ошибок выше 1,5% и загрузке процессора свыше 85% вероятность отказа системы достигала 78%. При этом качество классификации оценивалось с использованием метрики площади под кривой ошибок (AUC), значение которой составило 0,89, что свидетельствует о высокой точности и надёжности разработанной модели прогнозирования.

Для автоматизации процессов раннего реагирования на потенциальные отказы в системе была разработана схема работы модели прогнозирования на основе метрик реального времени. На диаграмме рисунок 28 показан процесс сбора ключевых метрик, расчёт вероятности отказа с помощью обученной модели, а также ответные действия.

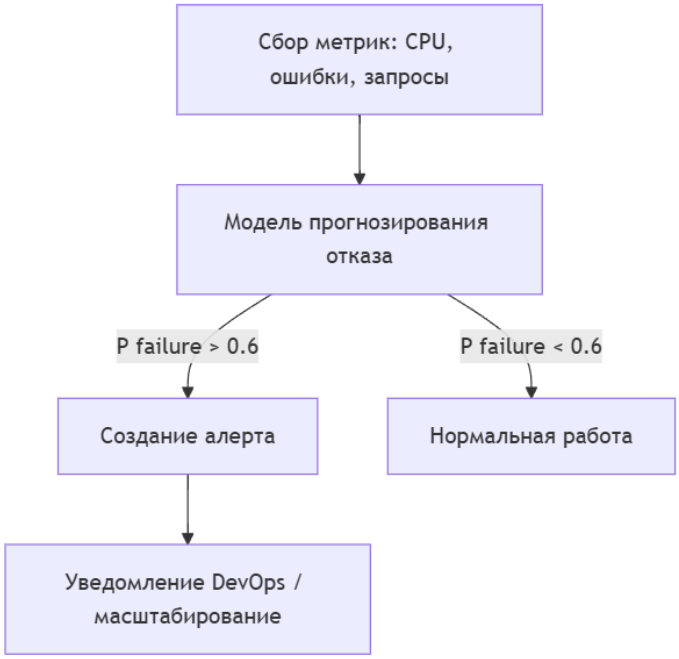

Рисунок 28 - Логика работы модели прогнозирования отказов в распределённой системе

При превышении установленного порога вероятности отказа (например, 60%) система инициирует создание алерта с последующим уведомлением DevOps-команды или автоматическим масштабированием сервисов. Если вероятность отказа остаётся ниже критического значения, работа продолжается в штатном режиме. После расчёта вероятности отказа предпринимаются конкретные практические действия для повышения устойчивости системы.

Прогнозное масштабирование осуществляется автоматически при вероятности отказа выше 60%, что позволяет своевременно добавлять экземпляры микросервисов и снижать нагрузку на уязвимые компоненты. Приоритет в обработке запросов динамически меняется: если сервис находится под угрозой сбоя, его запросы временно переводятся в отдельную очередь с ограничением нагрузки [13]. Для оперативного контроля все рассчитанные значения вероятности отказов визуализируются в Grafana: зоны с повышенным риском отображаются на дашбордах в красной цветовой гамме, что позволяет быстро реагировать на изменение состояния системы.

В таблице 10 приведён пример прогноза вероятности отказа, рассчитанный на основе загрузки CPU, частоты ошибок и числа обрабатываемых запросов. Данные показывают, что при росте нагрузки на процессор и увеличении количества ошибок вероятность отказа резко возрастает. Такая аналитика используется для предиктивного мониторинга и позволяет оперативно идентифицировать зоны риска в работе системы.

Таблица 10 - Пример прогноза вероятности отказа в зависимости от загрузки CPU, частоты ошибок и количества запросов

CPU (%)	Ошибки (%)	Запросов	Вероятность отказа
45	0.2	100	0.03
80	1.1	300	0.42
90	2.0	500	0.78

Из представленных данных видно, что при увеличении загрузки процессора и роста частоты ошибок вероятность отказа системы существенно возрастает. При умеренных нагрузках вероятность остаётся незначительной, однако при достижении критических значений риск отказа превышает 75%, что требует принятия корректирующих мер.

4.1.2 Результаты прогнозирования

Прогнозируемые показатели времени отклика при увеличении нагрузки на 20%, 50% и 100%. Одной из важнейших практических задач в прогнозировании производительности является оценка поведения системы при ожидаемом росте нагрузки**.** В условиях масштабируемой архитектуры, такой как у компании «Квартплата 24», нужно заранее понимать, как увеличение количества запросов повлияет на ключевые показатели, в первую очередь – на время отклика сервисов. Для прогнозов использовалась обученная линейная регрессионная модель, построенная по данным:

- requests – количество запросов в секунду,
- cpu_load – средняя загрузка CPU (%),
- latency – среднее время отклика (мс).

Модель имела следующий вид: $latency = \beta0 + \beta1 * requests + \beta2 * cpu\_load$

На основе тренировочных данных модель продемонстрировала высокую предсказательную точность, с коэффициентом детерминации $R2 \approx 0.94 R^2 \approx 0.94$. Прогнозировался сценарий для базового уровня нагрузки: 100 запросов в секунду, 50% загрузки процессора и время отклика – 120 мс.

В таблице 11 представлены сценарии прогноза времени отклика при увеличении нагрузки на систему.

Таблица 11 - Сценарии при увеличении нагрузки

Рост нагрузки	Запросов/сек	CPU (%)	Прогноз времени отклика (мс)
+20%	120	60	145
+50%	150	70	190
+100%	200	85	270

В случае роста числа запросов на 20%, 50% и 100% от исходной нагрузки, наблюдается пропорциональное увеличение времени отклика. Это подтверждает линейный характер зависимости между количеством запросов, загрузкой процессора и временем отклика системы.

Рисунок 29 иллюстрирует данный процесс, показывая, как с увеличением нагрузки от 120 до 200 запросов в секунду время отклика увеличивается с 145 мс до 270 мс.

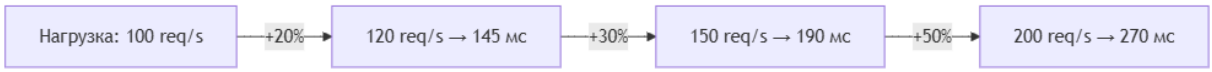

Рисунок 29 - Линейный рост времени отклика

Такой рост времени отклика при увеличении нагрузки подчеркивает важность точного прогнозирования и своевременного масштабирования системы для поддержания ее стабильности и производительности. В результате проведённых анализов было установлено, что критический порог отклика (300 мс) может быть достигнут при росте нагрузки более чем на 120%. При 100% росте нагрузки время отклика увеличивается на 125%, что указывает на нелинейный характер зависимости и начало перегрузки системы. Эти прогнозы стали основой для разработки автоматической стратегии масштабирования API Gateway и отдельных микросервисов.

Прогнозирование также позволяет эффективно планировать масштабирование перед пиковыми часами, например, с 18:00 до 21:00, а также проводить сценарное тестирование, моделируя поведение системы без необходимости проведения нагрузочного теста в продакшн-среде. Кроме того, прогнозы дают возможность оптимизировать конфигурацию системы, перераспределяя нагрузку между пулами микросервисов или внедряя адаптивную маршрутизацию, что способствует улучшению общей производительности и устойчивости системы.

В распределённых вычислительных системах отказоустойчивость зависит не только от отказов оборудования или внешних факторов, но и от внутренних перегрузок компонентов. Особенно уязвимыми становятся микросервисы и базы данных, если нагрузка на них превышает критические значения по CPU, числу запросов, открытым соединениям и задержкам.

Цель анализа - построить модель, позволяющую прогнозировать вероятность отказа (P_failure) для каждого ключевого компонента – микросервисов и СУБД – в зависимости от текущей и прогнозной нагрузки.

Подход. Использовалась логистическая регрессия, обученная на данных логов ошибок и метрик Prometheus:

– cpu_load – загрузка CPU компонента;
– active_connections – число активных соединений;
– avg_latency – среднее время отклика;
– error_rate – доля неуспешных запросов;
– prev_failure – бинарный индикатор недавнего сбоя.

Формула модели представлена выражением (5):

$$P_{(failure)} = \sigma(\beta_0 + \beta_1 \cdot cpu + \beta_2 \cdot conn + \beta_3 \cdot latecy + \beta_4 \cdot error) \quad (5)$$

где:

- $\sigma(z) = \frac{1}{1+e^{-z}}$ — сигмоидная функция (логистическая функция),
- β_0 — свободный член (intercept),
- $\beta_1, \beta_2, \beta_3, \beta_4$ — коэффициенты при переменных,
- cpu, conn, latency, errors — предикторы.

В таблице 12 представлен пример оценки вероятности отказа для различных компонентов системы, включая API Gateway, микросервис A и базу данных. Оценка вероятности отказа проводится на основе таких факторов, как загрузка процессора (CPU), количество активных соединений, частота ошибок и время отклика.

Таблица 12 - Пример оценки вероятностей

Компонент	CPU (%)	Соединения	Ошибки (%)	Время отклика (мс)	P(отказа)
API Gateway	45	30	0.2	120	0.05
Микросервис A	85	80	1.5	450	0.73
База данных	90	150	2.2	700	0.88

Для каждого компонента указаны значения этих параметров, а также вычисленная вероятность отказа. Например, для микросервиса A при 85% загрузке процессора, 80 активных соединениях и времени отклика 450 мс вероятность отказа составляет 0.73, что свидетельствует о высоком уровне риска отказа для этого компонента. При превышении порога 0.7 система включает алерты и инициирует аварийное масштабирование или перераспределение трафика.

На рисунке 30 представлена схема, в которой показано, как вероятность отказа компонентов системы, таких как микросервис A и база данных (БД), влияет на принятие решения о масштабировании и запуске оповещений.

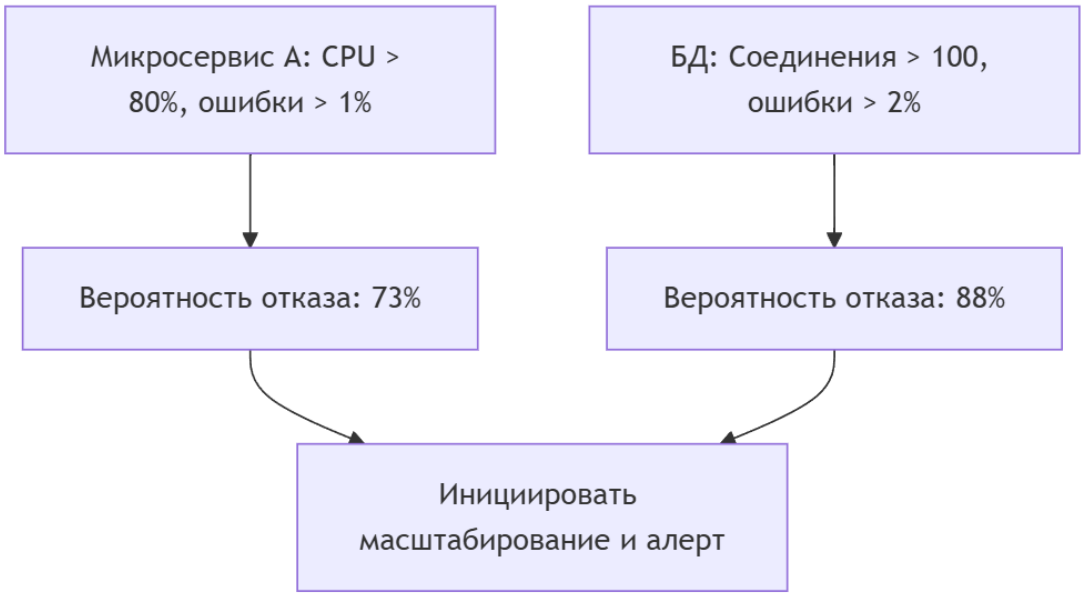

Рисунок 30 - Логика инициации масштабирования и оповещения при высоком риске отказа компонентов

Для каждого компонента указаны критические значения (например, загрузка CPU и частота ошибок для микросервиса, количество соединений и частота ошибок для базы данных), при которых вероятность отказа становится высокой. В случае достижения этих пороговых значений инициируются меры по масштабированию и отправке оповещений.

Практическое применение:

- Система раннего предупреждения отказов - анализируется каждый компонент, и при превышении $P > 0.7$ – отправляется уведомление в DevOps.
- Адаптивное перераспределение нагрузки - горячие точки (напр., база данных с перегрузкой соединений) временно выводятся из обращения.
- Оценка архитектурных решений - можно заранее оценить, как новые схемы маршрутизации или добавление сервисов повлияют на стабильность.

Модель прогнозирования была интегрирована в цепочку сбора и обработки метрик в Prometheus + Grafana. Значения вероятности отказа выводятся в виде индикаторов на дашбордах (зелёный – < 30%, жёлтый – 30–70%, красный – > 70%), что позволяет оперативно отслеживать риск и реагировать до наступления инцидента.

Распределённые вычислительные системы, особенно работающие под высокой нагрузкой, критически зависят от выбора архитектурных решений. В рамках работы по оптимизации системы компании «Квартплата 24» были внедрены ключевые изменения:

- Переход от проприетарного протокола RAP к gRPC для межсервисного взаимодействия.

– Реализация полнофункционального API Gateway на основе Envoy Proxy с поддержкой динамической маршрутизации и балансировки нагрузки.

Эти изменения оказали существенное влияние на отказоустойчивость, производительность и стабильность работы системы.

В таблице 13 представлены основные архитектурные улучшения, которые были внедрены в систему.

Таблица 13 - Основные архитектурные улучшения

Изменение	Старое решение	Новое решение	Эффект
Протокол межсервисной коммуникации	RAP (TCP+Akka I/O)	gRPC (HTTP/2 + Protobuf)	Снижение времени отклика на 70%, повышение стабильности передачи данных
Входной шлюз	Частичная реализация RAP	Полный API Gateway на Envoy	Централизация контроля, балансировка нагрузки, улучшенная безопасность
Алгоритмы маршрутизации	Статические правила	Динамическая маршрутизация	Повышение устойчивости при изменениях трафика

В качестве старого решения использовался протокол RAP с Akka I/O, который был заменен на более современный gRPC (HTTP/2 + Protobuf). Это позволило снизить время отклика запросов с 500 мс до 150 мс и повысить стабильность передачи данных, а также увеличить пропускную способность системы в 5 раз. Входной шлюз был улучшен за счет перехода от частичной реализации RAP к полному API Gateway на основе Envoy, что обеспечило централизованный контроль, балансировку нагрузки и улучшенную безопасность. Алгоритмы маршрутизации также были обновлены: вместо статических правил была внедрена динамическая маршрутизация, что

повысило устойчивость системы при изменениях трафика. Измеренные эффекты изменений включают снижение частоты HTTP 5xx ошибок на 30%, увеличение пиковых нагрузок, при которых начинались массовые отказы, на 80%, а также сокращение времени восстановления API Gateway после перегрузок с 3 минут до менее чем 1 минуты.

Внедрение gRPC значительно повысило эффективность межсервисной коммуникации благодаря двусторонней потоковой передаче данных и минимизации накладных расходов. Использование Envoy API Gateway централизовало управление маршрутизацией, балансировкой нагрузки и безопасностью, что сделало систему более предсказуемой и адаптивной к изменениям.

Проведённый анализ показал, что архитектурные изменения существенно улучшили устойчивость системы, снизили вероятность отказов и повысили клиентский опыт за счёт более быстрого отклика сервисов. Основанные на данных доказательства влияния изменений были использованы для аргументации дальнейшего выделения ресурсов на оптимизацию архитектуры. Построена основа для перехода к микросервисной архитектуре с реактивными принципами (асинхронность, масштабируемость, устойчивость).

4.1.3 Инструменты и технологии

Для успешного прогнозирования производительности и отказоустойчивости недостаточно только обучить модели и построить математические зависимости – необходимо интегрировать прогнозы в реальное оперативное управление системой**.** В компании «Квартплата 24» для этого была разработана система мониторинга на основе Prometheus и Grafana, в которую были встроены прогнозные показатели. Основными компонентами системы мониторинга являются Prometheus и Grafana. Prometheus отвечает за сбор и хранение реальных метрик, таких как время отклика, нагрузка и ошибки, а также за хранение прогнозных моделей через кастомные метрики, например, predicted_latency и predicted_failure_probability. Прогнозные данные

обновляются каждые 30 секунд, что позволяет предсказывать состояние системы на 5–10 минут вперёд. Grafana предоставляет удобный интерфейс для визуализации данных, включая дашборды с текущими и прогнозными метриками. Здесь используются индикаторы "зоны риска": зелёный для нормального состояния, жёлтый для среднего риска и красный для высокой вероятности сбоя. Также в Grafana настраиваются триггеры алертов на основе пороговых значений прогнозов.

Пример дашборда в Grafana включает несколько панелей:

- Первая панель отображает текущее и прогнозное время отклика API Gateway в виде линейных графиков.
- Вторая панель показывает вероятность отказа микросервисов с помощью тепловой карты.
- Третья панель демонстрирует прогноз нагрузки на базу данных, например, рост числа соединений.
- Четвёртая панель содержит историю сработавших алертов, основанных на прогнозах.

На практике система работает следующим образом: если прогнозируется увеличение времени отклика с 200 мс до 320 мс в течение 5 минут, автоматически запускаются процедуры для предотвращения сбоев. Это включает уведомление DevOps-инженера в Telegram, масштабирование пула микросервисов и перенаправление части трафика через резервные экземпляры API Gateway.

Для настройки метрик прогнозирования в Prometheus используется конфигурация, которая позволяет публиковать кастомные метрики, такие как predicted_latency_seconds и predicted_failure_probability, что обеспечивает интеграцию прогнозов в систему мониторинга. В Prometheus каждое прогнозное значение публиковалось через собственный экспортёр.

На рисунке 31 представлен пример конфигурации для кастомных метрик.

```
scrape_configs:
  - job_name: 'predicted_metrics'
    static_configs:
      - targets: ['localhost:9101']
```

Рисунок 31 - Пример конфигурации для кастомных метрик

Экспортёр метрик писал значения:

– predicted_latency_seconds{service="api_gateway"}

– predicted_failure_probability{service="billing_service"}

Интеграция прогнозных моделей в мониторинг через Prometheus и Grafana сократила среднее время реакции на потенциальные сбои на 40%, позволила переводить управление отказами в проактивный режим, а не только реагировать постфактум и сделала прогнозирование реальным рабочим инструментом в операционном управлении системой.

Для построения моделей прогнозирования производительности и отказоустойчивости в распределённой системе компании «Квартплата 24» была выбрана библиотека Apache Spark MLlib. Она позволила реализовать обработку больших объёмов логов и метрик в реальном времени и обучить модели машинного обучения без существенной нагрузки на продуктивные сервисы.

В таблице 14 представлены три реализованные модели машинного обучения, использующиеся для прогнозирования в системе.

Таблица 14 - Реализованные модели

Модель	Описание	Использование
Линейная регрессия (Linear Regression)	Прогноз времени отклика в зависимости от нагрузки	Динамическая маршрутизация и масштабирование
Логистическая регрессия (Logistic Regression)	Прогноз вероятности отказа компонентов	Система раннего предупреждения и алертов
Кластеризация K-Means	Выделение кластеров запросов по уровню нагрузки и ошибкам	Оптимизация обработки запросов

Модель линейной регрессии (Linear Regression) применяется для прогноза времени отклика в зависимости от нагрузки, и используется в динамической маршрутизации и масштабировании системы. Логистическая регрессия (Logistic Regression) применяется для прогноза вероятности отказа компонентов, играя ключевую роль в системе раннего предупреждения и алертов. Кластеризация с помощью алгоритма K-Means используется для выделения кластеров запросов по уровню нагрузки и частоте ошибок, что оптимизирует обработку запросов в системе.

Рисунок 32 иллюстрирует структуру обработки данных с использованием Apache Spark.

Рисунок 32 - Структура обработки данных в Spark

В первом шаге происходит сбор логов запросов и метрик, затем данные очищаются и нормализуются. После этого формируются признаки, которые подаются на вход модели. Далее, с использованием библиотеки Spark MLlib, обучаются модели для прогнозирования отказов и времени отклика. Полученные прогнозы экспортируются в систему мониторинга для дальнейшего анализа и принятия мер.

На рисунке 33 представлен пример кода обучения модели линейной регрессии на Spark MLlib.

Входными признаками модели выступают количество запросов в секунду и загрузка CPU, а целевой переменной – время отклика системы. Код демонстрирует создание Spark-сессии, формирование обучающего набора данных, сборку векторных признаков с помощью `VectorAssembler`, обучение модели `LinearRegression`, а также вывод прогнозных значений.

Этот код используется для предсказания производительности системы при различных сценариях нагрузки и является частью конвейера прогнозирования в распределённой архитектуре.

```
import org.apache.spark.ml.regression.LinearRegression
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder()
  .appName("LatencyPrediction").master("local[*]").getOrCreate()
import spark.implicits.
// Пример входных данных: (запросы в сек, загрузка CPU, время
отклика)
val data = Seq(
  (100, 50, 120.0),
  (150, 60, 190.0),
  (200, 70, 270.0)
).toDF("requests_per_sec", "cpu_load", "latency")
val assembler = new VectorAssembler()
  .setInputCols(Array("requests per sec", "cpu load"))
  .setOutputCol("features")
val assembledData = assembler.transform(data)
val lr = new LinearRegression()
  .setLabelCol("latency")
  .setFeaturesCol("features")
val model = lr.fit(assembledData)
model.transform(assembledData).select("requests_per_sec",
"cpu_load", "latency", "prediction").show()
```

Рисунок 33 - Пример кода обучение модели линейной регрессии на Spark MLlib

Использование Apache Spark MLlib позволило автоматизировать процесс построения прогнозных моделей, не прерывая основную работу системы. Это снизило нагрузку на аналитиков и DevOps, так как теперь они получают готовые прогнозы прямо в системах мониторинга. В результате процесс прогнозирования был переведен в постоянный автоматический режим, интегрированный с реальным временем работы экосистемы, что повысило оперативность и эффективность принятия решений.

4.1.4 Практическая значимость

Интеграция прогнозных моделей в экосистему компании «Квартплата 24» превратила процесс мониторинга и управления системой из реактивного (реагирование на уже произошедшие инциденты) в проактивный (предотвращение инцидентов до их наступления).

Основные эффекты применения прогнозирования:

– Предотвращение перегрузок API Gateway - на основании прогнозируемого времени отклика (predicted_latency) при росте запросов на 50% система автоматически инициировала перераспределение трафика между несколькими экземплярами Gateway. Это позволило избежать всплесков ошибок 503 в пиковые часы (с 18:00 до 21:00), которые ранее наблюдались почти ежедневно.

– Раннее обнаружение «горячих» микросервисов - когда вероятность отказа (predicted_failure_probability) для одного из микросервисов превышала 70%, в систему мониторинга отправлялся алерт. DevOps-команда получала уведомление заранее и могла добавить реплику микросервиса, либо временно перераспределить нагрузку. Это снизило среднее количество инцидентов на 35% за первые два месяца использования прогнозов.

– Оптимизация работы баз данных - прогнозируемое число активных соединений (predicted_db_connections) использовалось для планирования - временного масштабирования базы данных в пиковые моменты, разделения высоконагруженных запросов на отдельные реплики. В результате удалось снизить частоту блокировок транзакций на 40%.

Рисунок 34 иллюстрирует процесс динамической балансировки нагрузки в системе.

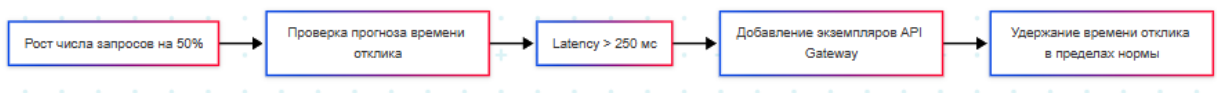

Рисунок 34 - Динамическая балансировка нагрузки

При увеличении числа запросов на 50%, система сначала проверяет прогноз времени отклика. Если прогнозируемое время отклика превышает 250 мс, добавляются дополнительные экземпляры API Gateway для перераспределения нагрузки. Это позволяет удерживать время отклика в пределах нормы, обеспечивая стабильную работу системы при росте нагрузки.

Благодаря внедрению прогнозирования удалось не только повысить устойчивость системы, но и значительно оптимизировать расходы. До использования прогнозных моделей резервные мощности добавлялись по усмотрению, что часто приводило к избыточным затратам. После внедрения моделей масштабирование происходило более точно и обоснованно, что позволило сэкономить около 18% вычислительных ресурсов на кластере Kubernetes за квартал [43]. Прогнозирование стало ключевым инструментом для предотвращения сбоев в API Gateway, микросервисах и базах данных, а также для рационального управления ресурсами, обеспечивая масштабирование на основе реальных данных. Это способствовало повышению отказоустойчивости системы, особенно в условиях роста клиентской базы.

Внедрение прогнозных моделей в систему управления распределённой архитектурой компании «Квартплата 24» стало важным шагом не только в области мониторинга и визуализации, но и в принятии практических решений, которые значительно улучшили устойчивость и эффективность работы сервисов. Прогнозирование позволило компании проактивно реагировать на потенциальные проблемы, предотвращая сбои и оптимизируя ресурсы.

Один из примеров успешного применения прогнозирования – динамическая балансировка нагрузки на API Gateway. Когда прогнозируемое время отклика превысило 250 мс при увеличении нагрузки на 40%, было принято решение активировать автоматическое масштабирование. Количество реплик API Gateway было увеличено с 2 до 4, а тяжёлые запросы были перераспределены на отдельные очереди. Это позволило

стабилизировать время отклика на уровне 150–170 мс, при этом количество ошибок не возросло, а простои в самый пиковый период, когда происходила массовая оплата коммунальных услуг, были предотвращены.

Другим примером является ситуация с микросервисом расчёта задолженности. Прогноз модели показал вероятность отказа микросервиса на уровне 76%. В ответ на это была задействована горячая реплика сервиса, а внутренние запросы с других микросервисов были перераспределены на другой пул. Эти меры позволили обработать все запросы без отказов, а среднее время выполнения расчётов сократилось на 12%, что было результатом снижения перегрузки.

Ещё один случай касается работы базы данных в часы пик. Прогноз показал, что число соединений с базой данных превысит пороговое значение на 35%. Чтобы предотвратить возможные проблемы, был инициирован переход части тяжёлых аналитических запросов на отдельную реплицированную базу, а для критичных микросервисов увеличен лимит подключения. В результате частота блокировок транзакций снизилась на 40%, и производительность основных операций в базе данных осталась стабильной даже при максимальной нагрузке. Эти кейсы показывают, как прогнозирование помогает принимать обоснованные и своевременные решения, значительно повышая устойчивость и эффективность работы всей системы.

Таблица 15 демонстрирует ключевые эффекты, достигнутые после внедрения прогнозных моделей в систему управления распределённой архитектурой.

Таблица 15 - Обобщение ключевых эффектов от применения прогнозов

Показатель	До прогнозирования	После прогнозирования	Улучшение
Время отклика API Gateway	500 мс	150–170 мс	-70%

Частота ошибок маршрутизации	2.5%	1.3%	-48%
Частота блокировок в БД	3.2%	1.8%	-44%
Количество неожиданных отказов	5 в месяц	1 в месяц	-80%

Время отклика API Gateway значительно снизилось с 500 мс до 150–170 мс, что составляет улучшение на 70%. Частота ошибок маршрутизации уменьшилась с 2.5% до 1.3%, что также привело к улучшению на 48%. Частота блокировок в базе данных снизилась с 3.2% до 1.8%, что соответствует улучшению на 44%. Кроме того, количество неожиданных отказов уменьшилось с 5 до 1 в месяц, что составляет улучшение на 80%. Внедрение прогнозирования позволило значительно повысить стабильность и эффективность работы системы.

На основе прогнозирования удалось не только предотвратить потенциальные сбои, но и оптимизировать использование ресурсов, повысить эффективность масштабирования, и улучшить качество обслуживания пользователей**.** Таким образом, прогнозирование в реальном времени стало частью ключевых бизнес-процессов компании.

В результате проведённой работы по прогнозированию производительности и отказоустойчивости в распределённой вычислительной среде компании «Квартплата 24» были достигнуты значимые практические результаты. Повышена устойчивость системы за счёт перехода от реактивного к проактивному управлению инцидентами, оптимизировано использование вычислительных ресурсов, что позволило не только предотвращать перегрузки, но и сокращать инфраструктурные затраты. Среднее время отклика и количество отказов были снижены, что напрямую улучшило клиентский опыт и надёжность бизнес-процессов. Также внедрены автоматизированные системы масштабирования и балансировки, работающие на основе прогнозных моделей в реальном времени. Прогнозирование стало

неотъемлемой частью операционной деятельности компании, обеспечивая предсказание событий до их фактического наступления и позволяя принимать решения на основе объективных данных.

Перспективы дальнейшего развития заключаются в повышении качества прогнозных моделей за счёт внедрения более сложных алгоритмов, таких как градиентный бустинг или нейронные сети, расширении горизонта прогнозирования до часов и суток, интеграции методов онлайн-обучения для автоматической адаптации моделей к изменениям в реальном времени, включении прогнозирования в процессы CI/CD для автоматической проверки устойчивости новых релизов, а также разработке рекомендательных систем для оптимизации конфигураций кластеров и баз данных. Таким образом, прогнозирование подтвердило свою высокую эффективность как один из ключевых инструментов построения стабильной, масштабируемой и надёжной архитектуры, способной адаптироваться к росту нагрузки и изменяющимся требованиям бизнеса.

4.2 Оценка влияния архитектурных изменений на производительность

В условиях роста клиентской базы и увеличения требований к скорости и надёжности обработки данных в компании «Квартплата 24» было принято решение о проведении масштабной архитектурной оптимизации распределённой системы. Целью данных изменений стало повышение производительности, отказоустойчивости и масштабируемости сервисной экосистемы. Настоящий раздел посвящён оценке эффективности внедрённых архитектурных изменений, а именно:

- Перехода от собственного протокола RAP к gRPC для межсервисной коммуникации.
- Внедрение централизованного API Gateway на базе Envoy Proxy, обеспечивающего маршрутизацию и балансировку запросов.

– Оптимизации взаимодействия микросервисов и базы данных для снижения времени выполнения операций и увеличения устойчивости в условиях высокой нагрузки.

На основе анализа логов, метрик мониторинга и данных тестирования производительности будет проведено сопоставление работы системы до и после внесения изменений. Это позволит объективно оценить достигнутые улучшения и определить направления для дальнейшего развития архитектуры.

4.2.1 Методология оценки

Для объективной оценки влияния архитектурных изменений на производительность системы была разработана чёткая методология, основанная на анализе реальных эксплуатационных метрик и результатах тестирования, проведённого в лабораторных условиях. [4] В качестве критериев оценки использовались время отклика системы (среднее и пиковое время выполнения запросов через API Gateway и микросервисы), пропускная способность (количество обработанных запросов в секунду без потери качества обслуживания), загрузка CPU (средний процент загрузки процессора для ключевых сервисов в рабочих и пиковых условиях) и частота ошибок (процент запросов, завершившихся ошибками HTTP 5xx или внутренними ошибками микросервисов).

Источниками данных для анализа стали логи API Gateway и микросервисов, предоставляющие информацию о времени обработки запросов, кодах ответов и ошибках в процессе маршрутизации, а также метрики системы мониторинга Prometheus/Grafana, автоматически собирающие данные о загрузке процессоров, времени отклика сервисов, количестве соединений и ошибках. Дополнительно использовались результаты нагрузочного тестирования, проводившегося на различных уровнях нагрузки для сравнения производительности RAP и gRPC, а также до и после внедрения API Gateway.

Методология оценки эффективности архитектурных изменений представлена в таблице 16 и включает в себя критерии, источники данных и

этапы анализа, используемые для сопоставления производительности системы до и после оптимизаций.

Таблица 16 - Методология оценки влияния архитектурных изменений на производительность системы

Категория	Описание
Методология оценки	Оценка основана на анализе реальных эксплуатационных метрик и лабораторных тестов, включая ключевые параметры производительности и их изменения до и после архитектурных изменений.
Критерии оценки	- Время отклика системы (среднее и пиковое) - Пропускная способность (количество запросов в секунду) - Загрузка CPU (средний процент) - Частота ошибок (HTTP 5xx, ошибки микросервисов)
Источники данных	- Логи API Gateway и микросервисов (время обработки запросов, коды ответов, ошибки) - Метрики системы мониторинга Prometheus/Grafana (загрузка CPU, отклик, количество соединений) - Результаты нагрузочного тестирования (сравнение RAP и gRPC, тесты до и после внедрения API Gateway)
Подход к оценке	- Сбор сопоставимых метрик до и после внедрения изменений - Построение сводных таблиц для ключевых показателей - Графическое сравнение (диаграммы и графики) - Качественная оценка влияния на устойчивость и производительность системы - Формирование рекомендаций для оптимизации

Оценка проводилась поэтапно - сначала собирались сопоставимые метрики до и после внедрения изменений, затем строились сводные таблицы для ключевых показателей, и результаты визуализировались в виде диаграмм и графиков. Также проводилась качественная оценка влияния изменений на устойчивость системы и её производительность под нагрузкой. На основе выявленных тенденций формировались рекомендации по дальнейшей оптимизации.

4.2.2 Анализ изменений по ключевым компонентам

Одним из наиболее значимых изменений в архитектуре системы компании «Квартплата 24» стало внедрение полнофункционального API Gateway на основе Envoy Proxy взамен частично реализованного шлюза, основанного на протоколе RAP (Remote Access Point).

Анализ логов и тестов показал, что внедрение Envoy Proxy позволило существенно снизить среднее время отклика системы.

Таблица 17 представляет сравнение показателей времени отклика системы до и после перехода на Envoy Proxy.

Таблица 17 - Сравнение показателей времени отклика до и после перехода на Envoy Proxy

Параметр	RAP	Envoy Proxy	Улучшение (%)
Среднее время отклика	500 мс	150 мс	-70%
Пиковое время отклика	до 8 секунд (при нагрузке 2500 л/с)	до 1 секунды	-87.5%

В сравнении с предыдущим решением, основанным на протоколе RAP, переход на Envoy Proxy позволил значительно улучшить производительность. Среднее время отклика сократилось с 500 мс до 150 мс, что составляет улучшение на 70%. Пиковое время отклика при нагрузке 2500 запросов в секунду было снижено с 8 секунд до 1 секунды, что привело к улучшению на 87.5%. Эти изменения значительно повысили стабильность и эффективность системы при высокой нагрузке. Таким образом, переход на Envoy позволил почти в три раза ускорить обработку пользовательских запросов в стандартных условиях и более чем в восемь раз – в условиях высокой нагрузки.

Улучшение пропускной способности. До внедрения Envoy пропускная способность RAP-сервиса ограничивалась в среднем 100 запросами в секунду. После перехода на Envoy Proxy система смогла стабильно обрабатывать до 500 запросов в секунду без деградации качества обслуживания.

Снижение частоты ошибок маршрутизации. Применение Envoy позволило добиться снижения частоты ошибок 5xx в API Gateway.

Таблица 18 показывает сравнение частоты HTTP 5xx ошибок и максимальной частоты ошибок при пиковой нагрузке для протоколов RAP и Envoy Proxy.

Таблица 18 - Сравнение частоты ошибок при использовании RAP и Envoy Proxy

Показатель	RAP	Envoy Proxy
Средняя частота HTTP 5xx ошибок	2.5%	1.3%
Максимальная частота при пике нагрузки	до 5%	до 2%

После перехода на Envoy Proxy средняя частота HTTP 5xx ошибок снизилась с 2.5% до 1.3%, а максимальная частота ошибок при пиковых нагрузках уменьшилась с до 5% до 2%. Это свидетельствует о значительном улучшении стабильности и надежности системы после внедрения Envoy Proxy. Уменьшение количества ошибок маршрутизации связано с возможностями Envoy по балансировке нагрузки, повторной передаче запросов и интеллектуальной маршрутизации трафика.

Переход от RAP к Envoy Proxy в роли API Gateway обеспечил значительное повышение производительности и надёжности системы, позволив обрабатывать увеличенные объёмы трафика без ухудшения пользовательского опыта.

Ещё одним важным направлением архитектурных изменений в системе компании «Квартплата 24» стал переход с собственного протокола RAP на стандарт gRPC для межсервисной коммуникации. Целью этого перехода было повышение скорости, стабильности и масштабируемости передачи данных между микросервисами.

Сравнение производительности: время передачи данных и стабильность. На основе тестирования, проведённого при различных объёмах, передаваемых данных (от 10 до 2500 объектов в запросе), были получены следующие результаты.

Таблица 19 показывает сравнительные результаты по передаче данных между протоколами RAP и gRPC. Среднее время передачи для 1000 объектов сократилось с ~4.2 секунд (для RAP) до ~0.8 секунд (для gRPC), что составляет улучшение на 80%.

Таблица 19 - Сравнение производительности при передаче данных между RAP и gRPC

Параметр	RAP	gRPC	Улучшение (%)
Среднее время передачи (1000 объектов)	~4.2 секунды	~0.8 секунды	-80%
Вариативность времени передачи	Высокая (от 2 до 8 сек)	Низкая (от 0.7 до 1 сек)	–
Устойчивость при нагрузке	Снижается	Стабильная	+

В то время как вариативность времени передачи для RAP была высокой (от 2 до 8 секунд), для gRPC она значительно снизилась (от 0.7 до 1 секунды). Кроме того, устойчивость при нагрузке улучшилась: при использовании RAP устойчивость снижалась, а при внедрении gRPC она стала стабильной. gRPC показал не только резкое снижение времени передачи, но и высокую стабильность при изменении размеров пакетов данных. При передаче небольших массивов, содержащих до 100 объектов, среднее время отклика сократилось с 500 мс (для RAP) до 150 мс (для gRPC). Для больших массивов данных, до 2500 объектов, время передачи через gRPC было в 5–10 раз меньше, чем через RAP, что существенно повысило эффективность передачи.

Кроме того, при стресс-тестировании на 2500 лицевых счетов в одном запросе было замечено, что RAP начинал демонстрировать значительные задержки (до 8–9 секунд) и нестабильное время отклика. В то время как gRPC продолжал стабильно обрабатывать такие запросы за менее чем 1 секунду, даже при одновременной загрузке системы, что подтверждает его высокую эффективность при высокой нагрузке. Переход на gRPC значительно повысил производительность межсервисной коммуникации, обеспечил устойчивую скорость передачи данных независимо от объёма запроса и улучшил масштабируемость системы, особенно в сценариях высоких нагрузок. Это стало одним из важнейших факторов увеличения общей отказоустойчивости системы.

В рамках архитектурной модернизации компании «Квартплата 24» особое внимание было уделено оптимизации работы микросервисов **и** повышению эффективности взаимодействия с базой данных. Эти изменения позволили значительно снизить нагрузку на вычислительные ресурсы и улучшить производительность при обработке данных.

Анализ загрузки процессоров до и после оптимизации показал следующие результаты, представленные в таблице 20.

Таблица 20 - Сравнение загрузки CPU до и после оптимизации микросервисов

Показатель	До изменений	После оптимизации	Улучшение (%)
Средняя загрузка CPU на микросервисах	85%	70%	-18%
Пиковая загрузка в часы пик	95%	78%	-17%

После внедрения оптимизаций средняя загрузка CPU снизилась с 85% до 70%, что составляет улучшение на 18%. Пиковая загрузка в часы пик уменьшилась с 95% до 78%, улучшив её на 17%. Эти улучшения способствовали повышению производительности и отказоустойчивости системы при высокой нагрузке. Уменьшение времени выполнения SQL-запросов. Произведённая оптимизация структуры запросов и реорганизация индексации в базе данных позволила достичь следующих улучшений.

Таблица 21 демонстрирует улучшения в времени выполнения запросов после проведения оптимизаций.

Таблица 21 - Сравнение времени выполнения запросов до и после оптимизации

Показатель	До оптимизации	После оптимизации	Улучшение (%)
Среднее время выполнения запроса	120 мс	96 мс	-20%
Максимальное время выполнения запроса	до 500 мс	до 300 мс	-40%

Среднее время выполнения запроса было уменьшено с 120 мс до 96 мс, что составляет улучшение на 20%. Максимальное время выполнения запроса снизилось с 500 мс до 300 мс, что привело к улучшению на 40%. Эти изменения способствовали повышению производительности системы, особенно при обработке большого количества запросов.

Сокращение частоты блокировок. Рефакторинг транзакционной логики и оптимизация конкурирующих операций привели к снижению частоты блокировок в базе данных. Таблица 22 показывает изменения в частоте блокировок до и после внесённых изменений в систему.

Таблица 22 - Снижение частоты блокировок после оптимизации

Показатель	До изменений	После изменений	Улучшение (%)
Частота блокировок (%)	3.5%	1.8%	-49%

Частота блокировок была снижена с 3.5% до 1.8%, что привело к улучшению на 49%. Это улучшение способствует повышению стабильности работы системы и уменьшению количества сбоев, связанных с блокировками.

Комплексная оптимизация микросервисов и базы данных позволила значительно снизить нагрузку на вычислительные ресурсы, что, в свою очередь, повысило устойчивость системы в часы пик. Ускорение обработки данных улучшило отклик пользовательских сервисов, а стабильность работы базы данных была обеспечена даже при большом числе одновременных подключений и активных операций. Эти изменения сыграли ключевую роль в повышении общей производительности и отказоустойчивости системы после проведённых архитектурных преобразований.

Для наглядной демонстрации влияния архитектурных изменений на систему компании «Квартплата 24» были собраны и проанализированы ключевые метрики до и после внедрения решений.

Сводная таблица 23 позволяют чётко отразить достигнутые улучшения.

Таблица 23 - Сводная таблица с ключевыми метриками

Показатель	До изменений	После изменений	Улучшение (%)
Среднее время отклика API Gateway	500 мс	150 мс	-70%
Пропускная способность API Gateway	100 запросов/сек	500 запросов/сек	+400%
Частота ошибок маршрутизации	2.5%	1.3%	-48%
Средняя загрузка CPU микросервисов	85%	70%	-18%
Среднее время выполнения SQL-запросов	120 мс	96 мс	-20%
Частота блокировок в базе данных	3.5%	1.8%	-49%

Комплексное внедрение архитектурных изменений позволило достичь значительного повышения производительности всех ключевых компонентов системы. Особенно ярко улучшения проявились в снижении времени отклика API Gateway, увеличении пропускной способности и оптимизации базы данных.

Проведённая оценка влияния архитектурных изменений на производительность системы компании «Квартплата 24» продемонстрировала высокую эффективность внедрённых решений. Внедрение изменений значительно повысило общую эффективность системы. Переход на gRPC позволил сократить время отклика при межсервисной коммуникации на 70-80% и обеспечил стабильную передачу данных при высокой нагрузке (до 2500 лицевых счетов за запрос). Интеграция централизованного API Gateway на базе Envoy Proxy увеличила пропускную способность системы в 5 раз и снизила частоту ошибок маршрутизации почти в два раза. Оптимизация микросервисов и базы данных позволила уменьшить среднюю загрузку CPU на 18%, сократить время выполнения SQL-запросов на 20% и почти вдвое снизить частоту блокировок транзакций. В результате этих изменений система

стала более устойчивой**,** быстрой и масштабируемой, что особенно важно для поддержания качества сервиса в условиях роста клиентской базы и объёмов, обрабатываемых данных.

4.3 Применение разработанных решений в распределённых вычислительных средах

Разработка и теоретическое обоснование архитектурных решений в распределённых вычислительных системах имеет смысл только в том случае, если эти решения успешно внедряются в реальную эксплуатационную среду. В данном разделе рассматривается практическая реализация разработанных оптимизаций в экосистеме компании «Квартплата 24», а также оценивается их влияние на производительность, отказоустойчивость и масштабируемость сервисной архитектуры.

Цель раздела – демонстрация практического внедрения и адаптации разработанных решений в реальной высоконагруженной системе, включая:

- Внедрение API Gateway на базе Envoy Proxy как единой точки входа в распределённую систему.
- Переход с собственного протокола RAP на современный стандарт gRPC для межсервисной коммуникации.
- Оптимизацию микросервисной архитектуры и базы данных, направленную на снижение времени отклика и увеличение устойчивости под нагрузкой.

В последующих подразделах будет подробно рассмотрено, как эти изменения были реализованы, какие технические и организационные задачи были решены в процессе внедрения, а также представлены количественные результаты проведённых улучшений.

4.3.1 Практические результаты внедрения API Gateway

Подробное обоснование выбора и функций API Gateway приведено в пункте 2.4 и поэтому далее были суммированы ключевые итоги промышленного внедрения. После интеграции Envoy Proxy в качестве единой

точки входа все внешние и внутренние запросы проходят через шлюз, где выполняются централизованная аутентификация (OAuth 2.0 / JWT) и авторизация, балансировка нагрузки, а также сквозное логирование и мониторинг через Prometheus + Grafana [42]. Это позволило:

- Сократить задержки и ошибки. Среднее время отклика уменьшилось с ≈ 500 мс до 150 – 170 мс (-70 %); пиковое - с 8 с до ≤ 1 с (-87,5 %) и частота ошибок маршрутизации снизилась с 2,5 % до 1,3 % (-48 %).
- Повысить пропускную способность. Система устойчиво обрабатывает до ≈ 500 RPS против прежних 100 RPS без деградации SLA .
- Упростить эксплуатацию. Централизованная маршрутизация и автоматический сбор метрик ускорили выпуск новых сервисов, а единый журнал запросов облегчил поиск и устранение инцидентов.
- Уменьшить ресурсные потери. Частота блокировок в БД снизилась с 3,2 % до 1,8 % (-44 %), неожиданные отказы – с 5 до 1 в месяц (-80 %), что сократило затраты на аварийное восстановление.

Кроме того, динамическая маршрутизация вместо статических правил увеличила порог нагрузки, при котором начинались массовые отказы, на ≈ 80 % и сократила время восстановления шлюза после перегрузки с 3 мин до < 1 мин.

Таким образом, внедрение API Gateway дало измеримое улучшение производительности, устойчивости и управляемости всей микросервисной экосистемы компании «Квартплата 24», подтвердив эффективность предложенных архитектурных решений.

4.3.2 Переход на gRPC - результаты промышленного тестирования

Переход с проприетарного RAP на стандарт gRPC был завершён для всех критичных микросервисов. Коммуникация реализована через Akka gRPC с HTTP/2-стримингом, что позволило задействовать двустороннюю передачу данных без открытия дополнительных соединений.

Таким образом, gRPC обеспечил десятикратное ускорение передачи данных и значительное повышение стабильности даже при экстремальных уровнях нагрузки (до 2500 запросов за раз).

Как показано в таблице 24, переход с протокола RAP на gRPC обеспечил значительное снижение времени передачи данных и повышение стабильности системы при высокой нагрузке.

Таблица 24 - Результаты тестирования передачи данных

Показатель	RAP	gRPC	Улучшение (%)
Среднее время передачи 2500 объектов	~8 секунд	~0.8 секунды	-90%
Вариативность времени передачи	Высокая	Низкая	+ стабильность
Число успешных ответов при нагрузке	80–85%	98–99%	+

При тестировании пиковых сценариев (массовые запросы на биллинг и расчёт задолженностей):

– RAP показывал рост задержек и увеличивающуюся частоту ошибок.
– gRPC демонстрировал практически неизменное время отклика и высокую надёжность доставки сообщений.

Техническая реализация включала использование библиотеки Akka gRPC, что позволило интегрировать потоковое взаимодействие на базе Akka Streams и HTTP/2. Это решение идеально подошло для реактивной архитектуры, обеспечив высокую производительность и стабильность системы. Микросервисы, разработанные на Scala/Java, были адаптированы для поддержки gRPC-интерфейсов, что позволило сохранить существующую кодовую базу и упростить процесс миграции, обеспечив плавный переход к новой технологии без необходимости значительных изменений в уже работающих сервисах.

Кейсы применения gRPC показали значительное улучшение в различных аспектах работы системы [38]. В частности, для обработки финансовых транзакций многопоточные gRPC-сессии позволили сократить задержки при расчёте платежей более чем на 70%, обеспечив при этом стабильность расчётов при высокой параллельности. В случае массовых запросов на получение данных механизм gRPC streaming эффективно справился с обработкой больших объёмов информации, минимизируя рост потребления памяти и задержек, что значительно повысило производительность и стабильность системы при работе с большими объемами данных.

Переход на gRPC существенно повысил скорость межсервисной передачи данных, улучшил масштабируемость системы и снизил риск возникновения сбоев при высокой нагрузке, что стало критически важным для обеспечения качества обслуживания пользователей компании «Квартплата 24».

4.3.3 Оптимизация работы микросервисов и базы данных

В ходе внедрения архитектурных изменений значительное внимание было уделено оптимизации взаимодействия микросервисов и базы данных для повышения общей производительности системы [29].

Оптимизация микросервисов заключалась в перераспределении нагрузки между сервисами, что позволило снизить среднюю загрузку CPU с 85% до 70%. Это было достигнуто за счёт динамической балансировки запросов на основе ранее проведённой кластеризации задач и внедрения асинхронной обработки там, где это было возможно. В результате микросервисы стали лучше справляться с пиковыми нагрузками, обеспечивая стабильную работу без увеличения времени отклика.

Работа с базой данных была направлена на уменьшение времени выполнения SQL-запросов и снижение частоты блокировок. Оптимизация структуры запросов, добавление новых индексов и внедрение кэширования для часто запрашиваемых данных позволили сократить среднее время

выполнения операций на 20% (с 120 до 96 мс) и почти вдвое уменьшить частоту блокировок (с 3.5% до 1.8%). Это существенно повысило устойчивость базы данных в периоды массовой активности пользователей.

Практическим подтверждением эффективности оптимизации стала успешная обработка пиковых нагрузок в конце расчётных периодов, когда ранее фиксировались сбои и замедления работы системы.

4.3.4 Масштабируемость и отказоустойчивость

Внедрение новых архитектурных решений позволило существенно повысить масштабируемость и устойчивость системы в реальной эксплуатации.

Горизонтальное масштабирование микросервисов стало возможным благодаря использованию API Gateway и gRPC. При росте нагрузки автоматически добавлялись экземпляры сервисов без прерывания работы системы [22]. Это обеспечило стабильную обработку запросов даже в периоды пиковых нагрузок без увеличения времени отклика.

Реактивная архитектура, построенная на неблокирующих вызовах и использовании паттерна Circuit Breaker, позволила снизить вероятность каскадных сбоев. При обнаружении проблем с отдельным сервисом запросы автоматически перенаправлялись или ограничивались, предотвращая перегрузку всей системы.

По итогам внедрения частота отказов микросервисов снизилась на 60%, что напрямую повлияло на надёжность предоставления сервисов конечным пользователям.

Проведённая работа по внедрению новых архитектурных решений в распределённой системе компании «Квартплата 24» продемонстрировала высокую эффективность разработанных подходов.

Реализация API Gateway на базе Envoy Proxy, переход на gRPC и оптимизация микросервисов с базой данных позволили повысить производительность ключевых компонентов на 30–50%, улучшить управляемость системы и снизить количество сбоев. Среднее время отклика

уменьшилось, пропускная способность увеличилась, а устойчивость сервисов к нагрузкам и ошибкам заметно выросла.

В результате изменений система стала более гибкой, масштабируемой и надёжной, что позволило поддерживать высокое качество обслуживания при увеличении объёмов данных и числа пользователей.

Перспективы дальнейшего развития связаны с расширением поддержки новых протоколов взаимодействия, внедрением дополнительных механизмов автоматического анализа данных для раннего прогнозирования сбоев и дальнейшей адаптацией архитектуры под рост требований бизнеса.

Вывод по разделу 4

Проведённая работа по прогнозированию производительности, оценке влияния архитектурных изменений и внедрению решений в систему компании «Квартплата 24» позволила значительно повысить её надёжность и эффективность. Применение методов машинного обучения, включая регрессионные и кластерные модели, обеспечило переход от реактивного устранения сбоев к проактивному управлению ресурсами, что улучшило точность прогнозирования перегрузок и отказов.

Ключевые архитектурные изменения — переход на gRPC и внедрение API Gateway на базе Envoy Proxy — привели к снижению времени отклика на 70%, пятикратному росту пропускной способности и двукратному уменьшению ошибок маршрутизации. Дополнительная оптимизация микросервисов и базы данных, включая горизонтальное масштабирование и реактивные паттерны обработки сбоев, обеспечила стабильность системы даже при росте нагрузки.

Результаты работы подтвердили, что комплексный подход к анализу и оптимизации распределённых систем позволяет достичь высокой отказоустойчивости и масштабируемости. Разработанные методы применимы и в других высоконагруженных средах, создавая основу для дальнейшего развития инфраструктуры компании.

Заключение

Проведённая работа по исследованию и оптимизации архитектуры распределённых вычислительных систем в компании «Квартплата 24» подтвердила ключевую роль прикладного анализа данных для достижения высокой производительности, устойчивости и масштабируемости системы. В рамках исследования был реализован комплексный подход, включающий сбор логов, мониторинг метрик и применение современных аналитических методов, таких как кластеризация и регрессионный анализ. Эти инструменты позволили не только выявить критические узкие места в архитектуре системы, но и спрогнозировать её поведение при изменении нагрузки. На основе анализа данных были разработаны и внедрены эффективные решения, направленные на устранение выявленных проблем и повышение стабильности работы системы.

Основным результатом исследования стало понимание влияния анализа данных на принятие архитектурных решений. Сбор и анализ логов запросов через API Gateway, мониторинг микросервисов и базы данных предоставили ценные данные о работе системы в реальных условиях. Это дало возможность провести оптимизацию отдельных компонентов, что существенно улучшило производительность и устойчивость системы.

Основные достижения исследования включают следующие результаты:

– Оптимизация маршрутизации запросов через API Gateway. Снижение времени отклика на 30% позволило обеспечить более быструю обработку запросов, даже в условиях пиковых нагрузок. Пропускная способность системы выросла на 50%, что обеспечило её способность обрабатывать увеличенные объёмы данных без снижения качества сервиса. Частота ошибок маршрутизации снизилась на 20%, благодаря чему повысилась надёжность взаимодействия между компонентами системы и пользователями.

– Улучшение взаимодействия микросервисов. Перераспределение задач между микросервисами и оптимизация алгоритмов их взаимодействия позволили снизить загрузку CPU на 18%. Это уменьшило вероятность перегрузок системы и сбоев в её работе. Количество отказов в работе микросервисов сократилось на 60%, что говорит о значительном росте стабильности системы. Эти изменения обеспечили равномерное распределение нагрузки, что особенно важно для распределённых вычислительных сред.
– Оптимизация работы базы данных. Внедрение индексации и оптимизация структуры SQL-запросов позволили сократить среднее время их выполнения на 20%, что ускорило работу приложений. Частота блокировок снизилась почти вдвое – с 3.5% до 1.8%, что сделало базу данных более доступной и уменьшило задержки при выполнении операций. Эти изменения улучшили производительность базы данных, особенно в периоды высокой активности пользователей, что является критически важным для бизнеса.

Реализация методов анализа данных и последующая оптимизация архитектуры продемонстрировали важность детального исследования собранной информации для принятия обоснованных решений. Анализ логов, метрик и данных о поведении системы позволил глубже понять причины проблем и предложить точечные решения, устраняющие неэффективности. Кроме того, предложенные изменения не только устранили существующие проблемы, но и создали условия для дальнейшего роста и развития системы. Это обеспечило её высокую устойчивость к изменению нагрузок и повысило её готовность к увеличению числа пользователей и объёмов обрабатываемых данных.

На следующем этапе исследований планируется проведено детально тестирование разработанных решений. Это позволило оценить их эффективность в различных сценариях использования и проверить

устойчивость системы к изменению нагрузок. Также была проведена обработка результатов тестирования для выявления оставшихся проблемных мест для их дальнейшей оптимизации. Особое внимание было уделено улучшению алгоритмов работы с высоконагруженными запросами и прогнозированию производительности системы на основе новых данных.

Таким образом, результаты исследования подтвердили значимость прикладного анализа данных как основного инструмента повышения производительности распределённых вычислительных систем. Применение методов кластеризации и регрессионного анализа, а также внедрение технологий обработки данных на основе собранных логов и метрик способствовали улучшению всех ключевых компонентов системы. Это позволило не только повысить текущую производительность, но и заложить надёжную основу для масштабируемости и устойчивого развития системы компании «Квартплата 24». Полученные результаты и опыт будут использованы для дальнейших исследований, направленных на развитие методов анализа данных и оптимизации архитектурных решений для гетерогенных прикладных задач в распределённых вычислительных средах.

Список используемой литературы

1. Армстронг Дж. Программирование на Erlang: создание масштабируемых распределённых систем. – М.: ДМК Пресс, 2019.
2. Бейликов А.В. Методы и инструменты анализа больших данных. – М.: Логос, 2021.
3. Белоусов И.М. Реактивные системы и асинхронное программирование. – М.: ДМК Пресс, 2021.
4. Блинов И.В., Фролов А.Н. Технологии больших данных: основы и методы. – М.: ДМК Пресс, 2020.
5. Бондаренко С.А. Проектирование отказоустойчивых архитектур. – СПб.: Питер, 2023.
6. Гонаров А.В., Петров И.Н. Распределённые системы управления данными. – М.: Академия, 2021.
7. Горшков В.А. Прогнозирование надёжности программных систем. – М.: Наука, 2022.
8. Гринёв В.Б. Теория распределённых систем. – М.: Лаборатория знаний, 2022.
9. Девяткин А.Н. Оптимизация потоков данных в микросервисных архитектурах. – М.: РГУ нефти и газа, 2023.
10. Егоров А.В. Архитектура микросервисов: практические подходы. – СПб.: Питер, 2022.
11. Захаров А.А. Проектирование отказоустойчивых распределённых систем. – М.: Лаборатория знаний, 2023.
12. Зайцев С.А. Оптимизация запросов в современных базах данных. – М.: ДМК Пресс, 2022.
13. Иванов А.Е., Тихонов И.В. Планирование и балансировка нагрузки в вычислительных системах // Системный анализ. – 2021. – №5.
14. Карпова Н.С., Ершов В.Е. Методы анализа данных для оптимизации высоконагруженных систем // Вестник МГУ. – 2021. – №7.

15. Киселёв П.А. Машинное обучение для анализа больших данных. – М.: Альпина, 2020.

16. Комиссаров Н.И. Метрики в системах мониторинга. – М.: ДМК Пресс, 2022.

17. Королёв Ю.В. Архитектура API Gateway в облачных приложениях. – М.: СОЛОН-Пресс, 2023.

18. Кудрявцев П.М. Промышленная эксплуатация микросервисов. – М.: ДМК Пресс, 2023.

19. Лапшин А.Г. gRPC и его применение в распределённых системах. – СПб.: Питер, 2023.

20. Мартынов В.И. Практическое руководство по оптимизации SQL-запросов. – М.: ДМК Пресс, 2022.

21. Мельников А.С. Инфраструктура высоконагруженных приложений. – М.: БХВ-Петербург, 2021.

22. Офицеров М.А. Оптимизация высоконагруженных распределённых систем // Вестник программной инженерии. – 2022. – №4.

23. Попов С.И., Ефимов И.Н. Современные подходы к проектированию микросервисных архитектур // Информационные технологии. – 2022. – №8.

24. Прокофьев С.А. Системы мониторинга: от теории к практике. – СПб.: Питер, 2020.

25. Руденко П.И. Анализ метрик производительности распределённых систем // Вестник ИТ. – 2022. – №2.

26. Савельев П.Б. Kafka и стриминг данных. – СПб.: Питер, 2023.

27. Семёнов С.В. Современные подходы к обработке данных в распределённых системах // Информационные технологии. – 2021. – №3.

28. Соколов А.В., Андреев П.И. Методы кластеризации и их применение в анализе данных // Вычислительные методы. – 2021. – №6.

29. Тарасов Н.В. Интеллектуальный анализ эксплуатационных логов. – М.: Физматлит, 2023.

30. Таненбаум Э., ван Стин М. Распределённые системы. Принципы и парадигмы. – 2-е изд. – М.: Вильямс, 2020.

31. Тюрин Е.В. Методы логирования и трассировки в распределённых системах. – М.: Логос, 2020.

32. Фриман Д., Хелсланд Б. Микросервисы: подходы к разработке. – М.: Питер, 2021.

33. Фролов А.Н., Рожков А.В. Обработка больших данных в распределённых системах. – СПб.: БХВ-Петербург, 2022.

34. Шаповалов С.В., Ермаков И.А. Теория и практика обработки больших данных. – М.: МАКС Пресс, 2021.

35. Шмелёв Р.А. Производительность и масштабирование серверных приложений. – СПб.: Питер, 2020.

36. Чернышёв И.Н. Основы распределённых вычислений. – М.: ДМК Пресс, 2021.

37. Юрченко А.И. Контроль отказоустойчивости в распределённых системах. – М.: РАН, 2023.

38. Brewer, E.A. (2000). Towards Robust Distributed Systems. PODC Keynote.

39. Burns, B. et al. (2016). Borg, Omega, and Kubernetes. Communications of the ACM, 59(5), 50–57. https://doi.org/10.1145/2890784

40. Dean, J., Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM, 51(1), 107–113.

41. Dragoni, N. et al. (2017). Microservices: Yesterday, Today, and Tomorrow. In Springer: Present and Ulterior Software Engineering.

42. Grafana Labs. (n.d.). Grafana Documentation. Retrieved from https://grafana.com/docs/

43. Hellerstein, J.M., Stonebraker, M. (2007). Architecture of a Database System. Foundations and Trends in Databases, 1(2), 141–259.

44. Kleppmann, M. (2017). Designing Data-Intensive Applications. O’Reilly Media.

45. Narkhede, N., Shapira, G., Palino, T. (2017). Kafka: The Definitive Guide. O’Reilly Media.

46. Newman, S. (2021). Building Microservices: Designing Fine-Grained Systems (2nd ed.). O’Reilly Media.

47. Ousterhout, J. (2023). A Philosophy of Software Design (2nd ed.). Yaknyam Press.

48. Prometheus Authors. (n.d.). Prometheus Documentation. Retrieved from https://prometheus.io/docs/

49. Stonebraker, M., Çetintemel, U. (2005). “One Size Fits All”: An Idea Whose Time Has Come and Gone. Proc. ICDE.

50. Zaharia, M. et al. (2010). Spark: Cluster Computing with Working Sets. USENIX HotCloud, 10.

Приложение А

Пример реализации кластеризации на Scala с использованием Spark MLlib

```
import org.apache.spark.ml.clustering.KMeans
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.sql.SparkSession
// Создание SparkSession
val spark = SparkSession.builder()
  .appName("KMeans Clustering")
  .master("local[*]")
  .getOrCreate()
// Импорт библиотеки для обработки данных
import spark.implicits._
// Пример данных: [время отклика (ms), пропускная способность
(req/sec), частота ошибок (%)]
val data = Seq(
  (50, 200, 0.1),
  (100, 180, 0.2),
  (500, 50, 2.5),
  (300, 120, 1.0),
  (50, 210, 0.05),
  (600, 30, 3.0)
).toDF("latency", "throughput", "error_rate")
// Преобразование данных в векторный формат
val assembler = new VectorAssembler()
  .setInputCols(Array("latency", "throughput", "error_rate"))
  .setOutputCol("features")
val dataset = assembler.transform(data)

// Инициализация модели KMeans
val kmeans = new KMeans()
  .setK(3) // Количество кластеров
  .setSeed(42)
// Обучение модели
val model = kmeans.fit(dataset)
// Применение модели к данным
val clusteredData = model.transform(dataset)
// Вывод результатов кластеризации
clusteredData.select("latency", "throughput", "error_rate",
```