МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

РАЗРАБОТКА МИКРОСЕРВИСНОГО ВЕБ-ПРИЛОЖЕНИЯ ДЛЯ РЕКОМЕНДАЦИИ ФИЛЬМОВ

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студента 5 курса 551 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Аркушина Святослава Владимировича

Научный руководитель
доцент, к. ф.-м. н. _____ А. С. Иванова

Заведующий кафедрой
к. ф.-м. н., доцент _____ С. В. Миронов

Саратов 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современный рост объёмов медиа-контента требует удобной навигации по фильмографическим базам и быстрого подбора релевантных фильмов. Микросервисный подход обеспечивает гибкость, масштабируемость и независимость компонентов, упрощая расширение и поддержку системы. В отличие от традиционных платформ с персонализированными рекомендациями на основе профиля пользователя, предлагаемое решение фокусируется на поиске сходства между фильмами: достаточно ввести название интересующего фильма, чтобы получить список максимально близких по содержанию или жанру. Это избавляет от необходимости собирать личные данные и настраивать сложные ML-модели, используя вместо этого метод k-ближайших соседей, реализованный в отдельном сервисе, который в реальном времени вычисляет расстояния в многомерном признаковом пространстве. Архитектура разделяет логику рекомендаций, веб-интерфейс, хранение данных и аутентификацию, что ускоряет разработку, упрощает масштабирование и позволяет внедрять новые алгоритмы без влияния на другие части системы.

Цель работы — спроектировать и разработать такое микросервисное веб-приложение, объединяющее преимущества распределённых сервисов и современных методов рекомендаций.

В ходе работы должны быть решены следующие задачи:

1. проанализировать технологический стек;
2. спроектировать архитектуру приложения;
3. реализовать пользовательскую часть приложения с использованием современных библиотек;
4. реализовать серверную часть веб-приложения с применением современных технологий;
5. реализовать сервис рекомендаций фильмов;
6. настроить и организовать общение между сервисами для обмена данными;
7. настроить быструю установку веб-приложения на сервер.

1 Анализ инструментов для разработки микросервисной рекомендательной системы

1.1 Современные рекомендательные приложения

Рекомендательные системы являются неотъемлемой частью цифровой экономики: они помогают пользователям быстро находить актуальный контент, повышая вовлечённость и время взаимодействия, что напрямую увеличивает доходы платформ. Сегодня такие механизмы применяются на «Кинопоиске», «Яндекс.Музыке», маркетплейсах и социальных сетях, где они персонализируют подборки фильмов, музыки, товаров и ленты публикаций.

Алгоритмы рекомендаций делятся на контентно-ориентированные, анализирующие свойства объектов; коллаборативную фильтрацию, основанную на поведении пользователей; и гибридные методы, объединяющие оба подхода для решения проблемы «холодного старта». Наряду с классическими методами всё более активно внедряются нейросетевые модели — глубокие автоэнкодеры, RNN, Transformer, графовые сети и обучение с подкреплением — что позволяет учитывать сложные нелинейные связи между пользователем, контентом и контекстом.

1.2 Пользовательская часть приложения

1.2.1 Инструменты сборки и разработки

Сборщики автоматизируют объединение, оптимизацию и трансформацию кода и ресурсов, ускоряя загрузку и разделяя бандлы по маршрутам приложения. Классические решения — Webpack с гибкой конфигурацией, Rollup с эффективным tree shaking для библиотек и Parcel с нулевыми настройками. Современные инструменты esbuild и Vite, основанные на нативных ES-модулях и движке на Go, предлагают мгновенный дев-сервер с «горячей» перезагрузкой и ультрабыстрый билд. Vite в режиме разработки подаёт исходники «как есть», трансформируя их по запросу, а в продакшене использует Rollup для tree shaking, минификации и генерации кастомизируемых бандлов без сложных конфигураций.

1.2.2 Библиотека React

React — декларативная JavaScript-библиотека для построения UI на основе изолированных компонентов, виртуального DOM и однонаправленного потока данных. С 2013 года она стала стандартом фронтенд-разработки: изменение

состояния автоматически обновляет лишь затронутые участки DOM, а Hooks (с React 16.8) позволяют функциональным компонентам управлять состоянием и эффектами без классов. Широкая экосистема (маршрутизаторы, формы, HTTP-клиенты), фреймворки Next.js и Remix для SSR/SSG и React Native для мобильных приложений делают React универсальным инструментом от простых SPA до масштабных микрофронтендов.

1.3 Серверная часть приложения

1.3.1 Библиотека Gin

Gin — минималистичный HTTP-фреймворк на Go, построенный вокруг сверхбыстрого маршрутизатора и цепочки middleware, что обеспечивает мгновенную обработку запросов и гибкое подключение логирования, аутентификации и других функций без дублирования кода. «Из коробки» он поддерживает автоматическую сериализацию Go-структур в JSON, XML и другие форматы, а механизм привязки ShouldBind выполняет валидацию входных данных по тегам. Не навязывая структуру проекта, Gin легко интегрируется с контекстами Go, WebSocket/SSE-расширениями и инструментами экосистемы (GORM, Ent, логгеры, профилировщики). Благодаря этой лёгкости и прозрачности он одинаково эффективен как в небольших микросервисах, так и в крупных распределённых системах, где критичны скорость отклика и надёжность.

1.4 Рекомендательная система

1.4.1 FastAPI и Uvicorn

FastAPI — современный асинхронный фреймворк на Python, который позволяет декларативно описывать HTTP-эндпоинты, автоматически валидировать входные данные и генерировать спецификацию OpenAPI «из коробки». Благодаря интеграции с asyncio и высокопроизводительному ASGI-серверу Uvicorn он демонстрирует минимальные задержки и высокую пропускную способность под нагрузкой. Единственным условием для надёжного использования в продакшене является тщательная настройка логирования и мониторинга, поскольку экосистема фреймворка ещё молода по сравнению с классическими WSGI-решениями.

1.4.2 Pandas и NumPy

Pandas и NumPy обеспечивают загрузку, объединение, фильтрацию и удаление дубликатов в табличных данных, а также ускоряют вычисления над числовыми массивами и преобразование признаков. При объёмах, превышающих размер оперативной памяти, целесообразно переходить к Spark или Dask подходам.

Для кодирования категориальных признаков и нормализации числовых столбцов используют scikit-learn: MultiLabelBinarizer и OneHotEncoder преобразуют жанры и типы, StandardScaler нормализует значения, а NearestNeighbors с косинусной метрикой обеспечивает поиск похожих объектов. В сценариях реального времени с миллионами векторов или при необходимости GPU-ускорения эффективнее применять Faiss или Annoy для быстрой индексации и поиска.

1.4.3 Pydantic

Pydantic в FastAPI отвечает за декларативное описание и строгую валидацию входящих и выходящих моделей данных, обеспечивая явные ошибки при нарушении формата и гарантируя корректность внутренних операций. При этом валидация больших массивов может быть ресурсоёмкой, поэтому в высоконагруженных сценариях её стоит профилировать или отключать на внутренних вызовах после фронтенд-валидации.

1.5 Брокеры сообщений

Брокеры сообщений обеспечивают асинхронную и надёжную коммуникацию между сервисами, позволяют буферизовать нагрузку, масштабировать продюсеров и потребителей независимо, подключать новые компоненты без изменения логики, а также поддерживать ретроспективный анализ и сложные workflow-паттерны. Однако использование брокеров влечёт за собой дополнительную сложность конфигурации и сопровождения, накладные расходы на задержки доставки, необходимость строгого контроля обработки ошибок и усложнение архитектуры при переходе на событийную модель взаимодействия.

1.5.1 Брокер сообщений Apache Kafka

Kafka обеспечивает надёжную доставку сообщений с минимальной задержкой, масштабируемость за счёт партиционирования, устойчивость к сбоям благодаря репликации, поддержку повторной обработки данных и гибкую

маршрутизацию событий, что делает её идеальной для построения высоконагруженных стриминговых систем. Вместе с тем её развёртывание и сопровождение требует глубоких знаний, избыточно для малых систем, упорядочивание сообщений возможно лишь в пределах одной партиции, а некорректная работа потребителей может привести к увеличению задержек и неконсистентной обработке.

1.5.2 Библиотека aiokafka

aiokafka — асинхронный клиент Kafka для Python, полностью построенный на asyncio, что позволяет обрабатывать тысячи сообщений без блокировки `event loop` в отличие от kafka-python. Библиотека поддерживает все ключевые механизмы Kafka (партиции, оффсеты, группы потребителей, балансировку), SSL/SASL-соединения и эффективно управляет буферизацией и переиспользованием соединений. Её нативная интеграция с FastAPI и другими async-фреймворками делает возможным одновременный приём, обработку и отправку сообщений параллельно с HTTP-запросами. Основные ограничения — повышенная сложность отладки асинхронного кода и относительно узкая экосистема примеров и документации, а также отсутствие высокоуровневой поддержки Kafka Streams.

1.5.3 Библиотека kafka-go

kafka-go предоставляет идиоматичный Go-интерфейс к Kafka без зависимости от C-библиотек, что упрощает деплой и компиляцию, особенно в облачных и контейнерных средах. Она позволяет вручную управлять продюсерами и консюмерами, задавать тонкие параметры буферов, партиционирования и оффсетов, использовать контексты Go для управления жизненным циклом запросов, а также легко собирать метрики и логи. Это делает её подходящей для кастомных решений в микросервисах, где требуется прозрачность и контроль. Однако по сравнению с confluent-kafka-go, использующей высокооптимизированную librdkafka, kafka-go может проигрывать в пиковых нагрузках. Кроме того, она ещё не поддерживает все расширенные возможности Kafka, такие как транзакции и идемпотентность, что ограничивает её применение в критически чувствительных сценариях с гарантированной доставкой без дубликатов.

1.6 Контейнеризация приложения (Docker)

Контейнеризация запускает приложения в изолированных средах с общим ядром хоста, упрощая деплой, масштабирование и сопровождение. Docker — популярная платформа, использующая образы (Dockerfile) и Registry (Docker Hub) для упаковки зависимостей. В микросервисной архитектуре фронтенд на React, бэкенды на Gin и FastAPI, Kafka-сервисы можно изолировать в контейнерах, объединить через docker-compose и запускать воспроизводимо как локально, так и в продакшене. Это упрощает локальную разработку, так как инфраструктура (Kafka, PostgreSQL, Redis) разворачивается одной командой.

Контейнеризация обеспечивает изоляцию зависимостей, воспроизводимость поведения приложений в любых средах, быстрый запуск, эффективное использование ресурсов, простую интеграцию в CI/CD-процессы и лёгкое масштабирование, особенно в сочетании с оркестраторами. Однако она требует дополнительного управления при большом количестве контейнеров, несёт потенциальные риски безопасности при работе с образами и может вызывать сетевые накладные расходы при интенсивной межсервисной коммуникации.

2 Разработка микросервисного приложения

2.1 Структура приложения

Приложение включает три сервиса — frontend (React/Vite), backend (Go с Gin и kafka-go) и recommendation-service (Python с aiokafka), объединённые через Apache Kafka и контейнеризированные с Docker Compose. Frontend отправляет запрос на backend, тот публикует его в Kafka-топик. Сервис рекомендаций обрабатывает запрос и отправляет ответ, который backend возвращает пользователю. Такая архитектура обеспечивает масштабируемость, отказоустойчивость и гибкость.

2.2 Разработка клиентской части приложения

Фронтенд построен на React с использованием React Router для навигации и хуков для управления состоянием. Интерфейс включает три маршрута: главную страницу поиска, страницу деталей фильма и страницу 404.

На главной странице пользователь вводит название фильма. Компонент `SearchInput` обрабатывает ввод с задержкой 300 мс и отображает подсказки. При выборе запускается асинхронная функция `fetchRecommendations`, и в `RecommendationsList` отображаются карточки фильмов.

Карточки содержат заголовок, год, тип, рейтинг и похожесть. Клик по карточке ведёт на маршрут `/movie/:imdbId`, где `MoviePage` загружает данные фильма по URL, отображая постер и информацию. Страница оформлена с помощью модульных SCSS, включая адаптивность и медиазапросы.

Вся логика организована реактивно и компонентно. Используется переменная окружения `VITE_API_URL` для обращения к бэкенду. Обработка ошибок предотвращает падения. Компоненты чётко разграничены: `HomePage` управляет состоянием, `SearchInput` — вводом, `SuggestionsList` — подсказками, `RecommendationsList` — выводом результатов.

Такой подход обеспечивает отзывчивый UX, предсказуемость и лёгкость поддержки.

2.3 Разработка серверной части приложения

Серверная часть реализована на Go с использованием Gin и модульной архитектурой, что обеспечивает масштабируемость и удобство поддержки. Основные функции — приём HTTP-запросов от фронтенда, взаимодействие с сервисом рекомендаций через Kafka, работа с локальной SQLite-базой и расшире-

ние данных через внешний API OMDb. При запуске инициализируется HTTP-сервер с CORS и регистрируются маршруты: автозаполнение названий, получение рекомендаций, детали фильма и проверка состояния. Каждый маршрут обрабатывается в отдельном модуле handler для ясности и разделения ответственности.

Автозаполнение реализовано через индексированные SQL-запросы с ограничением результата до 10, обеспечивая быструю отдачу и минимальную нагрузку. При запросе рекомендаций сервер формирует сообщение с уникальным `CorrelationID`, отправляет в Kafka и ждёт ответ с совпадающим идентификатором, возвращая ошибку 504 при таймауте. Для детальной информации сначала используется локальная база, затем внешний OMDb, данные объединяются в единый JSON; при отсутствии записи возвращается 404, при ошибках работы с базой — 500. Такая архитектура на базе модульности, асинхронности Kafka, локального кеша SQLite и расширения через OMDb обеспечивает гибкость, надёжность и масштабируемость сервера с предсказуемой обработкой ошибок и качественным логированием.

2.4 Разработка системы рекомендаций

Сервис рекомендаций реализован на FastAPI и aiokafka как асинхронное Python-приложение с фоновым Kafka-консюмером, который читает запросы с `correlation_id` и названием фильма из топика `recommendation_requests`. FastAPI предоставляет HTTP-эндпоинты для получения рекомендаций и проверки статуса, а при завершении корректно останавливает Kafka-консюмер и продюсер. Архитектура модульная: разделены загрузка данных, предобработка, модель, логика рекомендаций и API, что упрощает масштабирование и обновления без влияния на остальные компоненты.

Данные собираются из четырёх CSV-файлов с платформ Netflix, Apple TV, HBO Max и Amazon Prime, очищаются от дубликатов и неполных записей, объединяются в единую матрицу признаков с мультибинаризацией жанров, one-hot кодированием типа и нормализацией числовых параметров. Модель ближайших соседей на основе косинусного расстояния позволяет эффективно учитывать категориальные и числовые данные. При запросе сервис вычисляет коэффициенты похожести, исключая исходный фильм, и возвращает список похожих фильмов в топик `recommendation_responses`. Модульная структура обеспечивает гибкость, поддержку и лёгкое масштабирование сервиса.

2.5 Организация обмена данными через Apache Kafka

В микросервисе основой является асинхронный обмен сообщениями между HTTP-шлюзом и сервисом рекомендаций с использованием уникального `CorrelationID` — UUID, который связывает запросы и ответы и служит ключом Kafka. Kafka-сообщения с одним `ID` гарантированно попадают в одну партицию, сохраняя порядок, а метка времени помогает отладке. При проблемах с брокером применяется 5-секундный таймаут, после которого запрос завершается ошибкой. Backend слушает топик с таймаутом, предотвращая зависания, а Kafka-клиент автоматически фиксирует прочитанные сообщения, исключая дубли. Все операции логируются для мониторинга и диагностики.

При старте запускается FastAPI и фоновая корутина с Kafka-консюмером, чтобы не пропустить запросы после перезапуска. Каждое сообщение обрабатывается, вызывается ML-функция `recommend`, результаты сериализуются и отправляются асинхронным продюсером с гарантией записи. При остановке фоновые задачи отменяются, продюсер корректно закрывается, освобождая ресурсы. Архитектура опирается на асинхронность, явную инициализацию клиентов и корреляцию через `CorrelationID`, обеспечивая высокую производительность, устойчивость к сбоям и лёгкое масштабирование.

2.6 Контейнеризация приложения

Приложение контейнеризовано так, чтобы сервисы были изолированы, но взаимодействовали через Docker-сети и тома. В `docker-compose.yaml` описаны Kafka 3.8 в KRaft-режиме без ZooKeeper с сохранением данных в томе, сервис kafka-init для создания топиков с повторными попытками запуска, микросервис recommender на FastAPI с healthcheck и зависимостью от kafka-init, backend на Go/Gin с авто-перезапуском и frontend React/Vite с доступом к backend через переменную окружения. Все сервисы объединены в общей сети, что обеспечивает их взаимодействие по именам контейнеров, а браузеру — доступ по localhost. Такая архитектура даёт воспроизводимое окружение и лёгкое масштабирование, управляемое одной командой `docker-compose up --build`.

ЗАКЛЮЧЕНИЕ

В ходе выполнения бакалаврской работы была создана полностью модульная и масштабируемая система рекомендаций фильмов, основанная на сочетании современных подходов к разработке микросервисов, обработке данных и построению пользовательского интерфейса.

Для достижения поставленной цели в рамках работы:

1. были проанализирован технологический стек;
2. проектирована архитектура приложения;
3. реализована пользовательскую часть приложения с использованием современных библиотек;
4. реализована серверную часть веб-приложения с применением современных технологий;
5. реализован сервис рекомендаций фильмов;
6. настроено и организовано общение между сервисами для обмена данными;
7. настроена быстрая установка веб-приложения на сервер.

Разработанное веб-приложение представляет собо интуитивно понятный и привлекательный продукт, который способствует эффективному решению проблем пользователей.

Таким образом, работа не только достигла поставленных целей, но и продемонстрировала потенциал использования микросервисных приложения для улучшения удобства поиска фильмов конечных пользователей.