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

ВЕБ-ПРИЛОЖЕНИЕ ДЛЯ САМОРАЗВИТИЯ С ИНТЕЛЛЕКТУАЛЬНОЙ СИСТЕМОЙ АДАПТАЦИИ

БАКАЛАВРСКАЯ РАБОТА

студента 4 курса 451 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Стеклянникова Петра Сергеевича

Научный руководитель
доцент, к. ф.-м. н. _____ Сафрончик М.И.

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

Саратов 2026

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современное развитие цифровых технологий открывает новые возможности для создания адаптивных систем саморазвития и постановки целей. Такие системы призваны помочь пользователям эффективно планировать свой личностный рост, ставить достижимые цели и отслеживать прогресс в различных сферах жизни через геймифицированный подход.

Разрабатываемое веб-приложение представляет собой комплексную платформу для саморазвития, которая объединяет элементы геймификации, интеллектуальной персонализации и социального взаимодействия. Ключевой особенностью системы является ее адаптивность — способность подстраиваться под индивидуальные потребности и предпочтения каждого пользователя. Адаптивность достигается за счет двух взаимодополняющих механизмов: интеллектуальной системы рекомендаций на основе машинного обучения и автоматической генерации персонализированного контента через интеграцию с языковыми моделями.

Система рекомендаций использует современные алгоритмы машинного обучения (BERT) для анализа семантики контента и интересов пользователей, обеспечивая персонализированный подбор квестов и друзей с похожими целями. Генерация квестов через LLM API позволяет создавать уникальные задачи на основе текстовых запросов пользователей, адаптируя контент под их текущие потребности и интересы. Совместно эти механизмы обеспечивают динамическую адаптацию системы к каждому пользователю, создавая персонализированный опыт саморазвития.

Современные тенденции в разработке ПО, включая микросервисную архитектуру [1], прогресс в области машинного обучения и развитие облачных технологий, создают технологическую основу для реализации таких адаптивных систем. Микросервисный подход позволяет независимо масштабировать компоненты системы, а интеграция ML-сервисов обеспечивает интеллектуальную обработку данных для персонализации.

Практика посвящена исследованию, проектированию и разработке полнофункционального веб-приложения для саморазвития и постановки целей с интеллектуальной системой адаптации. В работе рассматриваются архитектурные решения для интеграции ML-сервисов, сравнивает-

ся эффективность различных алгоритмов рекомендаций, анализируются реализация системы генерации персонализированного контента.

Целью практики является исследование, проектирование и разработка адаптивного веб-приложения для саморазвития и постановки целей с интеллектуальной системой персонализации, представляющей собой полнофункциональную платформу с микросервисной архитектурой, интегрирующей современные алгоритмы машинного обучения (BERT) для рекомендаций и языковые модели для генерации персонализированного контента.

Для достижения поставленной цели в работе решается комплекс взаимосвязанных задач:

1. Проведение комплексного анализа предметной области, включая изучение современных подходов к адаптивным системам саморазвития, алгоритмов персонализации (Collaborative Filtering, Content-Based Filtering, BERT) и методов генерации контента через языковые модели.
2. Исследование и сравнительный анализ алгоритмов машинного обучения для персонализации: Collaborative Filtering на основе KNN, TF-IDF для текстового поиска, BERT для семантического анализа. Проведение сравнительного тестирования для выявления наиболее эффективного алгоритма.
3. Формирование и обоснование требований к разрабатываемой системе, включая функциональные требования (управление квестами и задачами, система рекомендаций, AI-генерация контента, социальные взаимодействия) и нефункциональные требования (производительность, точность рекомендаций, масштабируемость).
4. Проведение сравнительного анализа технологий и инструментов разработки для обоснованного выбора технологического стека, включающего средства реализации основного backend (Go, Gin), ML-сервиса (Python, FastAPI), интеграции с LLM API и системы хранения данных.
5. Проектирование архитектуры адаптивной системы на основе микросервисного подхода с определением границ сервисов, схем взаимодействия между основным backend (Go), Recommendation Service

(Python) и внешними сервисами (LLM API), моделей данных и API-интерфейсов.

6. Реализация алгоритма семантического поиска на основе BERT для поиска квестов с учетом семантического сходства текстовых описаний, преодолевающего ограничения ключевых слов.
7. Реализация алгоритма рекомендации квестов на основе BERT, использующего семантический анализ приобретенных квестов пользователя для генерации персонализированных рекомендаций.
8. Реализация алгоритма рекомендации друзей на основе BERT, анализирующего семантическое сходство интересов пользователей через BERT-эмбеддинги приобретенных квестов.
9. Разработка Recommendation Service на Python с использованием FastAPI, обеспечивающего RESTful API для взаимодействия с основным backend и эффективную обработку запросов на рекомендации.
10. Интеграция Recommendation Service с основным backend-приложением (Go), включая реализацию клиентских компонентов для взаимодействия с ML-сервисом, обработку ошибок и таймаутов.
11. Проведение сравнительного анализа эффективности различных алгоритмов (BERT vs TF-IDF vs Collaborative Filtering) на реальных данных, оценка метрик качества рекомендаций (precision, recall, F1-score).
12. Обеспечение производительности и масштабируемости Recommendation Service через оптимизацию алгоритмов, кэширование результатов и эффективную работу с данными.
13. Документирование процесса проектирования, разработки и развертывания системы рекомендаций, включая описание алгоритмов, архитектурных решений, API-документацию и результаты сравнительного анализа.

Решение указанных задач позволит создать полнофункциональный, технологически современный и практически значимый программный продукт, демонстрирующий комплексный подход к решению проблемы персонализации контента и рекомендаций на основе машинного обучения, а также подтверждающий высокий уровень профессиональной

подготовки автора работы в области программной инженерии и машинного обучения.

1 Анализ предметной области и существующих решений

1.1 Актуальность темы разработки адаптивных систем саморазвития

Разработка адаптивных систем для саморазвития и постановки целей представляет собой актуальное направление на стыке программной инженерии, машинного обучения и психологии мотивации. Актуальность темы обусловлена растущим интересом к цифровым инструментам для личностного роста и необходимостью создания систем, способных адаптироваться под индивидуальные потребности каждого пользователя.

Во-первых, наблюдается устойчивый рост спроса на цифровые платформы для саморазвития, которые помогают пользователям ставить цели, отслеживать прогресс и получать персонализированные рекомендации. Геймификация и системы мотивации показывают высокую эффективность в повышении вовлеченности пользователей и достижении целей. Персонализированные системы демонстрируют значительное преимущество по сравнению с универсальными решениями за счет адаптации под индивидуальные потребности каждого пользователя.

Во-вторых, адаптивность таких систем достигается за счет интеллектуальной персонализации контента. Современные алгоритмы машинного обучения, такие как BERT, позволяют анализировать семантику контента и интересы пользователей для генерации персонализированных рекомендаций. Дополнительно, интеграция с языковыми моделями (LLM) открывает возможности для автоматической генерации уникального контента, адаптированного под текущие потребности пользователя. Совместное использование этих механизмов создает динамически адаптирующуюся систему, которая подстраивается под каждого пользователя.

С технологической точки зрения актуальность темы подчеркивается созреванием необходимых технологических предпосылок: широкое распространение облачных вычислений, развитие микросервисных архитектур [1], прогресс в области обработки естественного языка и доступность предобученных моделей создают уникальную возможность для построения действительно интеллектуальных адаптивных систем.

1.2 Анализ современной литературы и существующих решений

1.2.1 Алгоритмы Collaborative Filtering

Collaborative Filtering является одним из наиболее распространенных подходов к построению систем рекомендаций. Основная идея метода заключается в использовании поведения и предпочтений других пользователей для предсказания интересов целевого пользователя. В работе Sarwar et al. (2001) [2] «Item-based collaborative filtering recommendation algorithms» подробно рассматриваются различные варианты Collaborative Filtering, включая user-based (поиск похожих пользователей) и item-based (поиск похожих элементов) подходы.

K-ближайших соседей (KNN) является классическим алгоритмом Collaborative Filtering, который находит пользователей или элементы, наиболее похожие на целевой объект, на основе метрик сходства (косинусное сходство, корреляция Пирсона — мера линейной зависимости между двумя переменными). Преимуществами KNN являются простота реализации и интерпретируемость результатов. Однако метод страдает от проблемы разреженности данных (sparsity problem) и холодного старта (cold start problem) для новых пользователей или элементов.

В работе Ricci et al. (2011) [3] «Recommender Systems Handbook» систематизированы различные подходы к Collaborative Filtering, включая матричную факторизацию и глубокое обучение.

1.2.2 Методы текстового поиска: TF-IDF

Term Frequency-Inverse Document Frequency (TF-IDF) [4,5] является классическим методом для оценки важности терминов в документах относительно коллекции документов. Формула TF-IDF определяется как:

$$TF\text{-}IDF(t, d) = TF(t, d) \times IDF(t)$$

где $TF(t, d)$ — частота термина t в документе d, а $IDF(t) = \log \frac{N}{df(t)}$ — обратная частота документа, где N — общее количество документов, а $df(t)$ — количество документов, содержащих термин t.

TF-IDF широко используется для текстового поиска и ранжирования документов. В контексте систем рекомендаций TF-IDF позволяет находить элементы с похожим текстовым содержанием, что особенно

полезно для рекомендации контента на основе описаний. Однако метод имеет ограничения: он не учитывает семантическое сходство и синонимию, работает только с точными совпадениями терминов.

1.2.3 Семантический анализ на основе BERT

Bidirectional Encoder Representations from Transformers (BERT) представляет собой модель трансформера [6], предобученную на больших корпусах текстов. BERT способен понимать контекст слов в обоих направлениях (bidirectional), что позволяет ему лучше улавливать семантические связи по сравнению с традиционными методами.

В работе Devlin et al. (2018) [7] «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding» показано, что BERT достигает state-of-the-art (наилучших на момент публикации) результатов в различных задачах обработки естественного языка. Для задач семантического поиска BERT [8] позволяет находить документы, семантически похожие на запрос, даже при отсутствии точных совпадений терминов.

Применение BERT для рекомендаций требует вычисления эмбеддингов текстовых описаний и использования метрик сходства (например, косинусное сходство) для поиска похожих элементов. Преимуществами BERT являются способность понимать семантику и контекст, что преодолевает ограничения TF-IDF. Однако метод требует значительных вычислительных ресурсов и времени на инференс (процесс применения обученной модели для получения предсказаний на новых данных).

1.2.4 Выбор алгоритма для продакшена

В рамках данной работы был проведен сравнительный анализ различных подходов к рекомендациям. Хотя теоретически гибридные методы могут показывать лучшие результаты, для данной системы было принято решение использовать только BERT. Это обусловлено несколькими практическими соображениями: во-первых, BERT демонстрирует достаточную точность для решения задач системы, во-вторых, использование единого алгоритма упрощает архитектуру и поддержку системы, в-третьих, это снижает вычислительную нагрузку по сравнению с комбинированием нескольких алгоритмов. Кроме того, BERT способен одновременно учитывать семантику контента и выявлять паттерны в данных

пользователей через семантические профили, что делает его универсальным решением для всех задач системы (поиск квестов, рекомендация квестов, рекомендация друзей). Также стоит отметить, что при тестировании гибридного подхода (комбинация BERT и Collaborative Filtering) результаты были немного лучше, но разница не оправдывала усложнение архитектуры системы.

1.2.5 Технологические аспекты разработки ML-сервисов

Современная литература по программной инженерии предлагает различные архитектурные подходы к построению систем машинного обучения. Работа Ричардсона (Richardson, 2018) [9] «Microservices Patterns» обосновывает преимущества микросервисной архитектуры для систем, требующих высокой масштабируемости и возможности независимого развертывания компонентов. Для ML-сервисов, которые могут требовать различных вычислительных ресурсов и частых обновлений моделей, такой подход представляется особенно подходящим [10].

FastAPI [11], современный веб-фреймворк для Python [12], обеспечивает высокую производительность и простоту разработки RESTful API для ML-сервисов. В работе Ramírez (2021) [13] «FastAPI Modern Python Web Development» рассматриваются практики разработки ML-сервисов с использованием FastAPI, включая асинхронную обработку запросов, валидацию данных и документацию API.

Вопросы производительности и масштабируемости ML-сервисов подробно освещены в работе Huyen (2022) [14] «Designing Machine Learning Systems», где рассматриваются методы оптимизации инференса, кэширования результатов и батчинга запросов (группировка нескольких запросов для одновременной обработки) для повышения пропускной способности системы.

1.2.6 Анализ существующих систем рекомендаций

Современные коммерческие системы рекомендаций (Amazon [15], Netflix [16], Spotify, YouTube) используют комбинации различных алгоритмов, включая Collaborative Filtering, матричную факторизацию и глубокое обучение. Однако детали реализации таких систем являются проприетарными и не раскрываются, что затрудняет воспроизведение ре-

зультатов и сравнительный анализ эффективности различных подходов. Открытые реализации, демонстрирующие сравнительный анализ эффективности различных методов (BERT, TF-IDF, Collaborative Filtering) в единой системе, встречаются редко, что определяет актуальность разработки системы с открытой реализацией и сравнительным анализом алгоритмов.

1.3 Технологические аспекты разработки систем рекомендаций

1.3.1 Архитектурные подходы и микросервисная архитектура

В контексте разработки систем рекомендаций особое значение приобретает выбор архитектурного подхода, обеспечивающего разделение основного backend и ML-сервиса. Микросервисная архитектура позволяет независимо разрабатывать, развертывать и масштабировать компоненты системы, что критически важно для ML-сервисов, требующих различных вычислительных ресурсов и частых обновлений моделей. Детальное описание выбранного технологического стека и обоснование выбора представлено в разделе "Технологический стек и обоснование выбора".

1.3.2 Библиотеки машинного обучения и обработки естественного языка

Для реализации алгоритмов рекомендаций используются следующие библиотеки Python:

sentence-transformers [17] — библиотека для работы с предобученными моделями трансформеров, включая BERT. Позволяет легко получать семантические эмбеддинги текстов для вычисления сходства между описаниями квестов. Используется многоязычная модель paraphrase-multilingual-MiniLM-L12-v2, оптимизированная для задач семантического поиска и поддерживающая работу с текстами на различных языках.

scikit-learn — библиотека для машинного обучения, используемая для реализации Collaborative Filtering на основе KNN. Предоставляет эффективные реализации алгоритмов ближайших соседей с различными метриками расстояния (косинусное сходство, евклидово расстояние).

TfidfVectorizer из scikit-learn — используется для реализации TF-IDF поиска. Обеспечивает векторизацию текстов с учетом частоты тер-

минов и обратной частоты документов, что позволяет эффективно находить релевантные квесты по текстовым запросам.

numpy и scipy — используются для эффективных вычислений с матрицами и векторами, необходимых для вычисления метрик сходства и обработки эмбеддингов.

1.3.3 Взаимодействие между микросервисами

Взаимодействие между основным backend (Go) и Recommendation Service (Python) осуществляется через RESTful API. Основной backend отправляет HTTP POST запросы к Recommendation Service с таймаутом 30 секунд для предотвращения зависаний при обработке запросов.

Структура взаимодействия:

- POST /api/search — поиск квестов по текстовому запросу (BERT)
- POST /api/quests/recommend — рекомендация квестов на основе истории пользователя (BERT)
- POST /api/users/recommend — рекомендация друзей на основе семантического сходства интересов (BERT)
- POST /api/quests/add — добавление новых квестов в индекс для поиска
- POST /api/users/add — добавление данных пользователей для Collaborative Filtering

Все запросы используют JSON для сериализации данных. Recommendation Service возвращает результаты с оценками сходства (similarity scores) и объяснениями рекомендаций, что позволяет пользователям понимать, почему им рекомендован тот или иной контент.

1.3.4 Кэширование и оптимизация производительности

Для обеспечения высокой производительности Recommendation Service используются техники кэширования эмбеддингов. Детальное описание реализованных оптимизаций представлено в разделе 2.7.5.

1.3.5 API design и REST principles

RESTful API проектируется в соответствии с принципами [18]:

- Stateless взаимодействие (без сохранения состояния) — каждый запрос содержит всю необходимую информацию для обработки, сер-

вер не хранит состояние сессии между запросами.

— Единообразие интерфейса — использование стандартных HTTP методов (GET, POST) и единообразной структуры URL для всех ресурсов, что упрощает понимание и использование API.
— Кэшируемость ответов — ответы сервера могут быть кэшированы клиентом для уменьшения нагрузки.
— Слоистая архитектура — разделение на слои, где каждый слой имеет четко определенную ответственность, что обеспечивает модульность и тестируемость системы.

Пример структуры endpoint'ов:

```
POST   /user/register              - регистрация
POST   /user/login                 - аутентификация
GET    /user/profile               - получение профиля
GET    /quests/available           - получение доступных квестов
POST   /quests/:questID/purchase   - покупка квеста
POST   /quests/:questID/start      - старт квеста
POST   /quests/:questID/:taskID/complete - выполнение задачи
GET    /friends                    - список друзей
POST   /friends/by-name/:friend_name - добавление друга по имени
```

1.3.6 Мониторинг и логирование

Для обеспечения наблюдаемости (observability) системы применяются:

— Разные уровни логирования (DEBUG, INFO, WARN, ERROR)
— Health checks и readiness probes

1.3.7 Тестирование и качество кода

Эти технологические аспекты формируют прочный фундамент для создания надежной, масштабируемой и эффективной системы рекомендаций, способной обрабатывать большие объемы данных и обеспечивать высокую точность рекомендаций.

2 Проектирование и реализация системы

2.1 Архитектура системы

Разрабатываемая система построена на микросервисной архитектуре с использованием технологического стека Go + Gin + PostgreSQL для основного backend, Python + FastAPI для Recommendation Service и HTML + CSS + JavaScript для frontend. Архитектура обеспечивает разделение ответственности между сервисами и возможность независимого масштабирования компонентов [1]. Общая архитектура системы представлена на рис. 1.

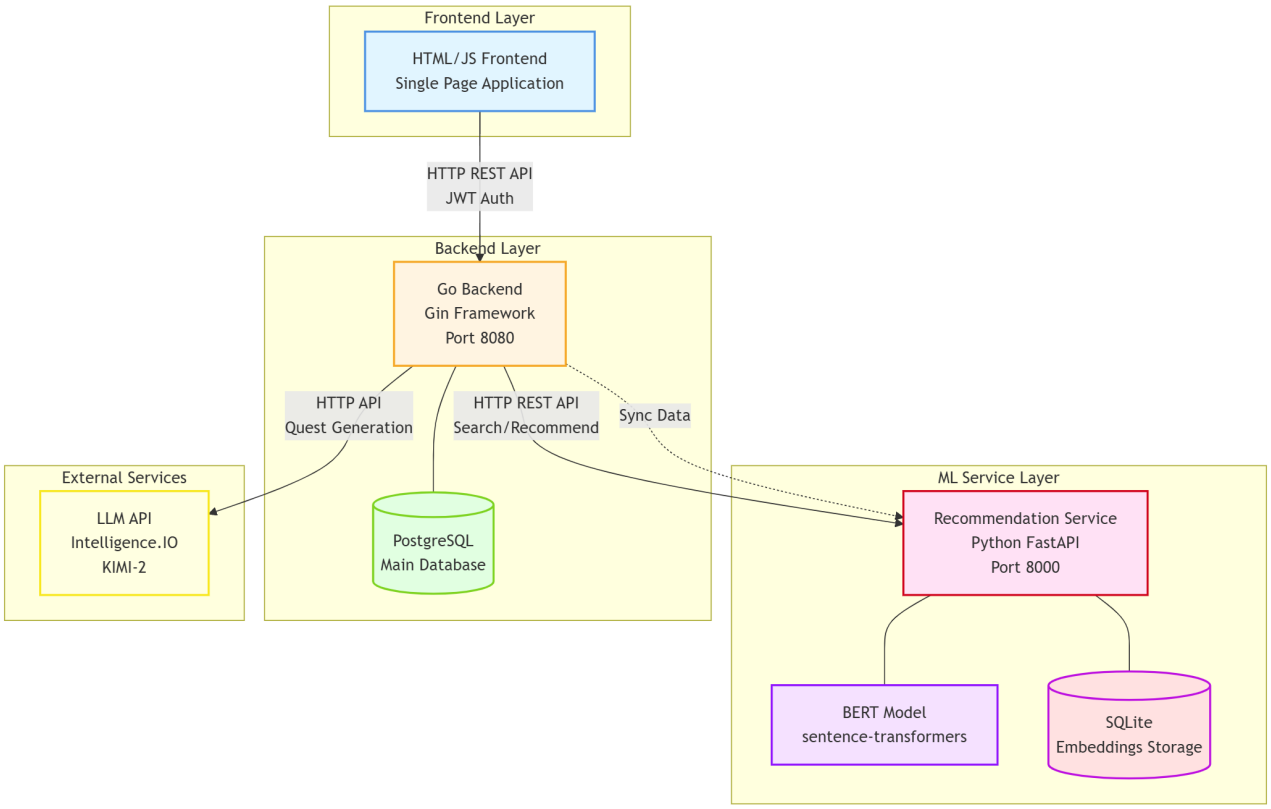

Рисунок 1 – Общая архитектура системы

Система организована в четыре основных слоя:

— Frontend Layer - веб-приложение на HTML и JavaScript, отвечающее за пользовательский интерфейс

— Backend Layer - Go сервер с фреймворком Gin, реализующий бизнес-логику и REST API

— ML Service Layer - Python сервис на FastAPI, реализующий алгоритмы рекомендаций и поиска на основе BERT (TF-IDF и Collaborative Filtering были протестированы для сравнения, но не используются

в продакшене)

— Data Layer - PostgreSQL база данных для хранения всей информации системы

Взаимодействие между Backend Layer и ML Service Layer осуществляется через HTTP REST API с таймаутами для обеспечения отказоустойчивости системы.

2.1.1 Технологический стек и обоснование выбора

Backend реализован на Go с использованием фреймворка Gin [19]. Выбор Go обусловлен его отличной поддержкой конкурентности, что критически важно для системы, обрабатывающей множество одновременных запросов к Recommendation Service [20].

Recommendation Service реализован на Python [12] с использованием FastAPI [11, 13]. Выбор Python обусловлен богатой экосистемой библиотек для машинного обучения (sentence-transformers [17] для BERT) и простотой интеграции предобученных моделей. В процессе разработки также использовались scikit-learn для реализации TF-IDF и Collaborative Filtering для сравнительного тестирования, но в продакшене используется только BERT.

База данных — PostgreSQL выбрана как надежная реляционная СУБД с богатым функционалом, поддерживающая сложные запросы и транзакции [21]. Используется для хранения метаданных квестов, истории взаимодействия пользователей и данных для построения семантических профилей.

База данных Recommendation Service — SQLite [22] используется для хранения BERT-эмбеддингов квестов и семантических профилей пользователей. Выбор SQLite обусловлен простотой развертывания (файловая база данных), достаточной производительностью для средних объемов данных и возможностью хранения бинарных данных (BLOB) для эмбеддингов. Данные сохраняются одновременно в SQLite для персистентности и в памяти для быстрого доступа, что обеспечивает высокую производительность при поиске и рекомендациях.

Frontend разработан на HTML [23] и JavaScript [24] без использования фреймворков. Выбор нативного подхода обусловлен простотой разработки и отсутствием зависимостей. Приложение использует современ-

ные библиотеки для специфических задач (FullCalendar для календаря, Cytoscape.js для визуализации графов квестов).

2.2 Реализация алгоритмов рекомендаций и поиска

В рамках разработки системы были протестированы различные алгоритмы машинного обучения для поиска и рекомендаций: TF-IDF, Collaborative Filtering (KNN) и BERT. Для оценки качества использовались метрики Precision@K, Recall@K, F1-score@K и NDCG@K. Сравнительный анализ на реальных данных показал превосходство BERT по всем метрикам качества, что в сочетании с его универсальностью (один алгоритм для всех задач) привело к выбору BERT как единственного алгоритма для продакшена.

2.2.1 Метрики оценки качества

Для оценки эффективности алгоритмов использовались следующие метрики:

- Precision@K — доля релевантных элементов среди top-K рекомендаций
- Recall@K — доля найденных релевантных элементов от общего количества релевантных
- F1-score@K — гармоническое среднее precision и recall
- NDCG@K — нормализованный дисконтированный кумулятивный выигрыш, учитывающий позицию релевантных элементов в ранжированном списке

2.2.2 Сравнительное тестирование: TF-IDF vs BERT

В процессе разработки был реализован и протестирован алгоритм TF-IDF для сравнения с BERT. TF-IDF (Term Frequency-Inverse Document Frequency) работает следующим образом:

1. Векторизация квестов: Каждый квест представляется в виде вектора TF-IDF на основе его названия, описания и категории. Используется TfidfVectorizer из scikit-learn с параметром максимального количества признаков: 1000.
2. Векторизация запроса: Пользовательский запрос преобразуется в вектор TF-IDF с использованием той же модели векторизации.

3. Вычисление сходства: Косинусное сходство между вектором запроса и векторами всех квестов вычисляется по формуле:

$$\text{similarity}(q, d) = \frac{\mathbf{q} \cdot \mathbf{d}}{||\mathbf{q}|| \times ||\mathbf{d}||}$$

где $\mathbf{q}$ — вектор запроса, $\mathbf{d}$ — вектор документа (квеста).

4. Ранжирование и фильтрация: Квесты ранжируются по убыванию сходства, применяется фильтрация по категориям и статусу (если указано), возвращаются top-K наиболее релевантных результатов.

Преимущества TF-IDF: быстрая обработка запросов (O(n) где n — количество квестов), низкие требования к вычислительным ресурсам, хорошая интерпретируемость результатов.

Ограничения: не учитывает семантическое сходство (синонимы, контекст), работает только с точными совпадениями терминов, требует предобработки текста (токенизация, стемминг).

Результаты тестирования (рекомендации квестов):

Алгоритм	Precision@5	Recall@5	F1@5	NDCG@5
TF-IDF	0.480	0.306	0.370	0.482
Collaborative Filtering (KNN, k=3)	0.400	0.246	0.303	0.464
BERT	0.540	0.338	0.412	0.569

Таблица 1 – Сравнение алгоритмов для рекомендации квестов

BERT продемонстрировал более высокие результаты по всем метрикам по сравнению с TF-IDF и Collaborative Filtering. Хотя разница незначительна, BERT показал лучшее понимание семантики и контекста, что важно для работы с текстовыми описаниями квестов. В связи с этим для продакшена был выбран BERT.

2.2.3 Алгоритм семантического поиска на основе BERT (основной метод)

BERT (Bidirectional Encoder Representations from Transformers) [7, 8] используется для семантического поиска квестов, преодолевающего ограничения ключевых слов. Алгоритм реализован следующим образом:

1. Предобработка текстов: Названия и описания квестов объединяются в единый текст, который нормализуется (удаление лишних пробелов, приведение к нижнему регистру).
2. Генерация эмбеддингов: Используется предобученная многоязычная модель paraphrase-multilingual-MiniLM-L12-v2 из библиотеки sentence-transformers [17]. Модель генерирует 384-мерные эмбеддинги для каждого квеста и поддерживает работу с текстами на различных языках:

$$\mathbf{e}_d = \text{BERT}(\text{concat}(\text{title}_d, \text{description}_d))$$

   где $\mathbf{e}_d$ — эмбеддинг квеста d.
3. Кэширование эмбеддингов: Эмбеддинги всех квестов предвычисляются при загрузке сервиса и сохраняются в памяти для быстрого доступа, что позволяет избежать повторных вычислений при каждом запросе.
4. Семантический поиск: При получении поискового запроса генерируется эмбеддинг запроса $\mathbf{e}_q$, затем вычисляется косинусное сходство со всеми эмбеддингами квестов:

$$\text{similarity}(q, d) = \cos(\mathbf{e}_q, \mathbf{e}_d) = \frac{\mathbf{e}_q \cdot \mathbf{e}_d}{||\mathbf{e}_q|| \times ||\mathbf{e}_d||}$$

5. Ранжирование: Квесты ранжируются по убыванию семантического сходства, возвращаются top-K наиболее релевантных результатов.

Преимущества BERT: понимание семантики и контекста, способность находить релевантные результаты даже при отсутствии точных совпадений терминов, учет синонимов и связанных понятий, превосходство по всем метрикам качества по сравнению с TF-IDF.

Ограничения: более высокие требования к вычислительным ресурсам, большее время обработки запросов по сравнению с TF-IDF, необходимость предобученных моделей. Однако улучшение качества рекомендаций компенсирует увеличение времени обработки.

Результаты тестирования BERT для поиска квестов представлены в таблице 1. На основе этих результатов BERT выбран как основной и единственный алгоритм для поиска квестов в продакшене.

2.2.4 Сравнительное тестирование: Collaborative Filtering (KNN) vs BERT

Также в процессе разработки был реализован и протестирован алгоритм Collaborative Filtering на основе KNN для сравнения с BERT. Collaborative Filtering работает следующим образом:

1. Построение матрицы взаимодействий: Создается бинарная матрица $M_{n \times m}$, где n — количество пользователей, m — количество квестов. Элемент $M_{i,j}$ представляет наличие квеста у пользователя:

$$M_{i,j} = \begin{cases} 1 & \text{если пользователь } i \text{ добавил квест } j \text{ себе} \\ 0 & \text{иначе} \end{cases}$$

2. Вычисление сходства пользователей: Для целевого пользователя u вычисляется сходство со всеми другими пользователями с использованием косинусного сходства:

$$\text{similarity}(u, v) = \frac{\mathbf{M}_u \cdot \mathbf{M}_v}{||\mathbf{M}_u|| \times ||\mathbf{M}_v||}$$

   где $\mathbf{M}_u$ и $\mathbf{M}_v$ — векторы взаимодействий пользователей u и v.
3. Поиск K ближайших соседей: Используется алгоритм KNN из scikit-learn (NearestNeighbors) с метрикой косинусного сходства для нахождения K наиболее похожих пользователей (по умолчанию K=3).
4. Генерация рекомендаций: Для каждого квеста, который не был добавлен целевым пользователем, вычисляется сумма весов похожих пользователей, у которых есть этот квест:

$$\text{score}(u, q) = \sum_{v \in N(u)} w(u, v) \times \mathbf{1}_{M_{v,q}=1}$$

   где $N(u)$ — множество K ближайших соседей пользователя u, $w(u, v) = 1 - d(u, v)$ — вес на основе косинусного расстояния $d(u, v)$ между пользователями, $\mathbf{1}_{M_{v,q}=1}$ — индикатор наличия квеста q у пользователя v.
5. Ранжирование и фильтрация: Квесты ранжируются по убыванию score, применяется фильтрация по категориям (если указано), воз-

вращаются top-K наиболее релевантных рекомендаций.

Преимущества Collaborative Filtering: учет поведенческих паттернов пользователей, способность находить неочевидные связи между квестами, не требует анализа содержания квестов.

Ограничения: проблема холодного старта для новых пользователей или квестов, проблема разреженности данных при малом количестве взаимодействий, вычислительная сложность $O(n^2)$ для больших пользовательских баз.

Результаты тестирования Collaborative Filtering представлены в таблице 1. BERT превосходит Collaborative Filtering по всем метрикам. В связи с этим для продакшена выбран BERT как единственный алгоритм для рекомендаций.

2.2.5 Использование BERT для рекомендаций квестов

На основе результатов сравнительного тестирования для рекомендаций квестов и друзей используется семантический анализ на основе BERT. Алгоритм работает следующим образом:

1. Построение семантического профиля интересов пользователя: На основе приобретенных квестов пользователя формируется семантический профиль через BERT-эмбеддинги приобретенных квестов. Профиль представляет собой усредненный эмбеддинг всех приобретенных квестов:
$$\mathbf{p}_u = \frac{1}{|U|} \sum_{q \in U} \mathbf{e}_q$$
где U — множество приобретенных квестов пользователя u, $\mathbf{e}_q$ — BERT-эмбеддинг квеста q.
2. Семантическое ранжирование квестов: Для каждого потенциально рекомендованного квеста вычисляется семантическое сходство с профилем пользователя:
$$\text{score}(u, q) = \cos(\mathbf{p}_u, \mathbf{e}_q)$$
где $\mathbf{e}_q$ — BERT-эмбеддинг квеста q.
3. Фильтрация и ранжирование: Квесты фильтруются по категориям (если указано) и ранжируются по убыванию score, возвращают-

ся top-K рекомендаций с объяснениями на основе семантического сходства.

Преимущества использования BERT для рекомендаций: глубокое понимание семантики контента, способность находить релевантные квесты даже при отсутствии точных совпадений, учет контекста и связанных понятий, превосходство по всем метрикам качества по сравнению с Collaborative Filtering.

2.2.6 Использование BERT для рекомендаций друзей

Рекомендация друзей основана на семантическом анализе интересов пользователей с использованием BERT:

1. Построение семантического профиля интересов пользователя: Для каждого пользователя создается семантический профиль на основе BERT-эмбеддингов приобретенных квестов. Профиль представляет собой усредненный эмбеддинг всех приобретенных квестов:

$$\mathbf{p}_u = \frac{1}{|U|} \sum_{q \in U} \mathbf{e}_q$$

   где U — множество приобретенных квестов пользователя u, $\mathbf{e}_q$ — BERT-эмбеддинг квеста q.
2. Вычисление семантического сходства интересов: Для целевого пользователя вычисляется косинусное сходство его семантического профиля со всеми другими пользователями:

$$\text{similarity}(u, v) = \cos(\mathbf{p}_u, \mathbf{p}_v)$$

   где $\mathbf{p}_u$ и $\mathbf{p}_v$ — семантические профили пользователей.
3. Ранжирование пользователей: Пользователи ранжируются по убыванию семантического сходства.
4. Фильтрация существующих друзей: Из списка рекомендаций исключаются пользователи, с которыми уже установлена дружба.
5. Генерация объяснений: Для каждой рекомендации генерируется объяснение на основе семантического сходства интересов и общих категорий приобретенных квестов.

Алгоритм	Precision@3	Recall@3	F1@3
TF-IDF	0.633	0.658	0.636
Collaborative Filtering (KNN, k=3)	0.633	0.650	0.631
BERT	0.633	0.700	0.641

Таблица 2 – Сравнение алгоритмов для рекомендации друзей

Все три алгоритма показали схожие результаты по precision, но BERT продемонстрировал лучший recall (0.700), что означает способность находить больше релевантных друзей. Это важно для социальных функций системы. Кроме того, использование единого алгоритма (BERT) для всех задач системы (поиск квестов, рекомендация квестов, рекомендация друзей) упрощает архитектуру и поддержку.

Преимущества использования BERT для рекомендации друзей: глубокое понимание семантики интересов пользователей, способность находить друзей с похожими интересами даже при отсутствии точных совпадений в категориях, учет контекста и связанных понятий в выполненных квестах.

2.2.7 Общие выводы по сравнительному анализу

BERT демонстрирует лучшие результаты по всем метрикам по сравнению с TF-IDF и Collaborative Filtering. Особенно заметно преимущество по NDCG@5 для рекомендаций квестов (0.569 против 0.482 у TF-IDF и 0.464 у Collaborative Filtering), что говорит о лучшем ранжировании результатов. Хотя разница в F1-score незначительна, BERT лучше понимает семантику текстовых описаний квестов [25, 26], что важно для качества рекомендаций. На основе этих результатов BERT был выбран как единственный алгоритм для всех задач системы в продакшене.

2.3 Реализация основного backend-приложения

Основной backend-сервис реализован на Go с использованием фреймворка Gin и обеспечивает всю бизнес-логику приложения, взаимодействие с базой данных PostgreSQL и интеграцию с внешними сервисами (LLM API и Recommendation Service).

2.3.1 Система аутентификации и авторизации

Реализована полнофункциональная система аутентификации на основе JWT (JSON Web Tokens). Процесс аутентификации включает следующие этапы:

1. Регистрация пользователя: При регистрации пароль хешируется с использованием алгоритма bcrypt с параметром bcrypt.DefaultCost (равен 10), что обеспечивает защиту от brute-force атак. Хеш пароля сохраняется в таблице users вместе с уникальными username и email.
2. Аутентификация: При успешной аутентификации генерируется JWT токен с временем жизни 24 часа. Токен содержит идентификатор пользователя (user_id) в payload и подписывается секретным ключом, хранящимся в переменных окружения.
3. Авторизация: Middleware JWTAuthMiddleware проверяет наличие и валидность токена в заголовке Authorization каждого защищенного запроса. При успешной проверке идентификатор пользователя сохраняется в контексте запроса для дальнейшего использования в обработчиках.

Кодовая реализация системы аутентификации представлена в приложении Б. Полная реализация слоистой архитектуры backend-приложения, включая примеры обработчиков, сервисов и репозиториев, представлена в приложении Д.

2.3.2 Система уровней и геймификация

Реализована система прогрессии игрока на основе накопленного опыта (XP) с использованием квадратичной формулы расчета уровня:

$$level = \lfloor \sqrt{\frac{XP}{100}} \rfloor + 1$$

Данная формула обеспечивает экспоненциальный рост требований к опыту для каждого следующего уровня, что создает сбалансированную прогрессию:

— Уровень 1: 0-99 XP
— Уровень 2: 100-399 XP
— Уровень 3: 400-899 XP

— Уровень 4: 900-1599 XP

— И так далее

При выполнении задачи или завершении квеста опыт начисляется пользователю, и уровень автоматически пересчитывается с использованием функции calculateLevel. Обновление уровня происходит атомарно в рамках транзакции вместе с начислением опыта и монет, что гарантирует целостность данных.

Реализация функции calculateLevel:

```
// calculateLevel вычисляет уровень игрока на основе опыта
// Использует квадратичную прогрессию (quadratic progression)
// Формула: level = floor(sqrt(XP / base)) + 1
// Где base = 100 - базовое значение для балансировки прогрессии
func calculateLevel(xp int) int {
    if xp < 0 {
        xp = 0
    }

    const baseXP = 100.0

    // Квадратичная формула: level = floor(sqrt(XP / base)) + 1
    level := int(math.Floor(math.Sqrt(float64(xp) / baseXP))) + 1

    // Минимальный уровень = 1
    if level < 1 {
        level = 1
    }

    return level
}
```

2.3.3 Система фильтрации квестов по уровню сложности

Квесты фильтруются на основе уровня пользователя для обеспечения постепенного открытия контента. Алгоритм фильтрации реализован в методе GetAvailableQuests:

1. Получение текущего уровня пользователя и баланса монет из базы данных.
2. Фильтрация квестов по следующим критериям:

— difficulty <= user.level + 1 — пользователь может видеть квесты со сложностью не более чем на 1 уровень выше его текущего уровня
— price <= user.coin_balance — пользователь может видеть только те квесты, которые он может себе позволить
— Квест не должен быть уже куплен или пройден пользователем

3. Возврат отфильтрованного списка доступных квестов.

Такая система обеспечивает постепенное открытие нового контента по мере прогрессии пользователя, создавая мотивацию для выполнения квестов и повышения уровня.

Реализация метода GetAvailableQuests:

```
func (r *QuestRepository) GetAvailableQuests(
    ctx context.Context,
    userID int,
) ([]models.Quest, error) {
    // Получаем уровень пользователя и баланс
    var user models.User
    err := r.db.GetContext(ctx, &user,
        "SELECT * FROM users WHERE id = $1", userID)
    if err != nil {
        return nil, err
    }

    // Фильтруем квесты по уровню и балансу
    query := `
        SELECT q.* FROM quests q
        WHERE q.difficulty <= $1 + 1
          AND q.price <= $2
          AND NOT EXISTS (
            SELECT 1 FROM user_quests uq
            WHERE uq.quest_id = q.id AND uq.user_id = $3
          )
    `

    var quests []models.Quest
    err = r.db.SelectContext(ctx, &quests, query,
        user.Level, user.CoinBalance, userID)
    if err != nil {
        return nil, err
```

```
29     }
30
31     return quests, nil
32 }
```

2.3.4 Управление квестами и задачами с использованием транзакций

Все критические операции с квестами и задачами выполняются в рамках транзакций PostgreSQL для обеспечения атомарности и целостности данных. Используется метод BeginTxx для создания транзакций с контекстом, что позволяет корректно обрабатывать таймауты и отмену операций.

Процесс покупки квеста реализован следующим образом:

1. Начало транзакции: tx, err := r.db.BeginTxx(ctx, nil)
2. Проверка существования квеста и достаточности средств пользователя
3. Создание записи в user_quests со статусом 'purchased'
4. Создание записей в user_tasks для всех задач квеста со статусом 'not_started'
5. Списывание монет с баланса пользователя: UPDATE users SET coin_balance = coin_balance - price
6. Создание записи транзакции в user_coin_transactions для аудита финансовых операций
7. Подтверждение транзакции: tx.Commit() или откат при ошибке: tx.Rollback()

Процесс выполнения задачи также выполняется в транзакции:

1. Проверка существования квеста в статусе 'started' или 'purchased'
2. Проверка существования задачи в статусе 'active' или 'not_started'
3. Если квест в статусе 'purchased', автоматический старт квеста и активация всех задач
4. Получение базовых наград задачи (base_xp_reward, base_coin_reward)
5. Начисление наград пользователю с автоматическим пересчетом уровня через функцию addXPAndCoinsWithLevelUp
6. Обновление статуса задачи на 'completed' с сохранением начисленных наград

7. Подтверждение транзакции

Использование транзакций гарантирует, что все связанные операции выполняются атомарно: либо все изменения применяются, либо ни одно из них не применяется, что критически важно для финансовых операций и системы прогрессии.

Реализация транзакционного начисления наград методом addXPAndCoinsWith

```
func (r *QuestRepository) addXPAndCoinsWithLevelUp(
    tx *sqlx.Tx,
    ctx context.Context,
    userID, xpAmount, coinAmount int,
) error {
    // Получаем текущий опыт пользователя
    var currentXP int
    err := tx.GetContext(ctx, &currentXP,
        "SELECT xp_points FROM users WHERE id = $1", userID)
    if err != nil {
        return err
    }

    // Вычисляем новый опыт и уровень
    newXP := currentXP + xpAmount
    newLevel := calculateLevel(newXP)

    // Начисляем опыт, монеты и обновляем уровень атомарно
    _, err = tx.ExecContext(ctx, `
        UPDATE users
        SET xp_points = xp_points + $1,
            coin_balance = coin_balance + $2,
            level = $3
        WHERE id = $4`,
        xpAmount, coinAmount, newLevel, userID)
    if err != nil {
        return err
    }

    return nil
}
```

Реализация транзакционной обработки покупки квеста методом PurchaseQuest представлена в приложении Д. Ниже приведен краткий пример ключе-

вой логики:

```
func (r *QuestRepository) PurchaseQuest(
    ctx context.Context,
    userID, questID int,
) error {
    tx, err := r.db.BeginTxx(ctx, nil)
    if err != nil {
        return err
    }
    defer tx.Rollback()

    // Получаем информацию о квесте
    var quest models.Quest
    err = tx.GetContext(ctx, &quest,
        "SELECT * FROM quests WHERE id = $1", questID)
    if err != nil {
        return err
    }

    // Проверяем баланс пользователя
    var balance int
    err = tx.GetContext(ctx, &balance,
        "SELECT coin_balance FROM users WHERE id = $1", userID)
    if err != nil {
        return err
    }

    if balance < quest.Price {
        return errors.New("insufficient funds")
    }

    // Создаем запись в user_quests
    _, err = tx.ExecContext(ctx, `
        INSERT INTO user_quests (user_id, quest_id, status)
        VALUES ($1, $2, 'purchased')`,
        userID, questID)
    if err != nil {
        return err
    }

    // Создаем user_tasks для всех задач квеста
```

```
41      _, err = tx.ExecContext(ctx, `
42          INSERT INTO user_tasks (user_id, task_id, quest_id, status)
43          SELECT $1, qt.task_id, qt.quest_id, 'not_started'
44          FROM quest_tasks qt
45          WHERE qt.quest_id = $2
46          ORDER BY qt.task_order
47      `, userID, questID)
48      if err != nil {
49          return err
50      }
51
52      // Списываем монеты
53      _, err = tx.ExecContext(ctx, `
54          UPDATE users SET coin_balance = coin_balance - $1 WHERE id = $2`,
55          quest.Price, userID)
56      if err != nil {
57          return err
58      }
59
60      // Записываем транзакцию для аудита
61      _, err = tx.ExecContext(ctx, `
62          INSERT INTO user_coin_transactions
63          (user_id, amount, transaction_type, reference_type, reference_id, description)
64          VALUES ($1, $2, 'spent', 'quest', $3, 'Purchased quest: ' || $4)`,
65          userID, -quest.Price, quest.ID, quest.Title)
66      if err != nil {
67          return err
68      }
69
70      return tx.Commit()
71  }
```

2.3.5 AI-генерация квестов через LLM API

Реализована интеграция с LLM API (Intelligence.IO Solutions) для генерации персонализированных квестов на основе текстового запроса пользователя. Используется модель moonshotai/Kimi-K2-Thinking, которая обладает встроенной системой цензуры для предотвращения генерации вредного контента.

Процесс генерации квеста:

1. Получение запроса: Пользователь отправляет текстовый промпт с

описанием желаемого квеста через endpoint POST /quests/generate.

2. Формирование системного промпта: Создается детальный системный промпт, определяющий структуру JSON-ответа, включающую метаданные квеста (название, описание, категория, редкость, сложность, цена, награды) и массив задач с их параметрами.
3. Отправка запроса к LLM: Выполняется HTTP POST запрос к API https://api.intelligence.io.solutions/api/v1/chat/completions с использованием Bearer токена для аутентификации. Параметры запроса:
   — Model: moonshotai/Kimi-K2-Thinking
   — Temperature: 0.7 (баланс между креативностью и детерминированностью)
   — Messages: системный промпт и пользовательский запрос
4. Обработка ответа: LLM возвращает JSON-структуру с квестом и задачами. Ответ очищается от thinking-тегов модели (модель использует формат thinking для внутренних рассуждений).
5. Парсинг и валидация: JSON-ответ парсится в структуру AIQuestResponse, содержащую объект Quest и массив Tasks. Выполняется валидация структуры данных.
6. Сохранение в базу данных: Сгенерированный квест и задачи сохраняются в базу данных через метод SaveQuestToDB, который:
   — Вставляет запись в таблицу quests
   — Вставляет записи в таблицу tasks
   — Создает связи в таблице quest_tasks
   — Возвращает идентификатор созданного квеста
7. Интеграция с Recommendation Service: В фоновом режиме (goroutine) отправляется асинхронный запрос к Recommendation Service для добавления нового квеста в индекс поиска и рекомендаций. Это позволяет сразу использовать сгенерированный квест в системе рекомендаций.
8. Возврат результата: Пользователю возвращается полная информация о сгенерированном квесте, включая его идентификатор, метаданные и список задач.

Внутренняя цензура KIMI 2: Модель moonshotai/Kimi-K2-Thinking обладает встроенной системой фильтрации контента, которая автомати-

чески блокирует генерацию квестов с вредным, опасным или неподходящим содержанием. Проведенные тесты показали эффективность данной системы цензуры в предотвращении генерации нежелательного контента.

Реализация интеграции с LLM API методом requestAI (полная версия представлена в приложении В):

```
func requestAI(userMessage, systemPrompt, aiModel string) ([]byte, error) {
    if apiKey == "" {
        return nil, fmt.Errorf("API_KEY not found")
    }

    if aiModel == "" {
        aiModel = "moonshotai/Kimi-K2-Thinking"
    }

    requestData := ChatRequest{
        Model: aiModel,
        Messages: []ChatMessage{
            {Role: "system", Content: systemPrompt},
            {Role: "user", Content: userMessage},
        },
        Temperature: 0.7,
    }

    jsonData, err := json.Marshal(requestData)
    if err != nil {
        return nil, fmt.Errorf("error marshaling JSON: %v", err)
    }

    req, err := http.NewRequest("POST", url, bytes.NewBuffer(jsonData))
    if err != nil {
        return nil, fmt.Errorf("error creating request: %v", err)
    }

    req.Header.Set("Content-Type", "application/json")
    req.Header.Set("Authorization", "Bearer "+apiKey)

    client := &http.Client{Timeout: 30 * time.Second}
    resp, err := client.Do(req)
    if err != nil {
```

```
35          return nil, fmt.Errorf("error sending request: %v", err)
36      }
37      defer resp.Body.Close()
38
39      body, err := io.ReadAll(resp.Body)
40      if err != nil {
41          return nil, fmt.Errorf("error reading response: %v", err)
42      }
43
44      if resp.StatusCode != http.StatusOK {
45          return nil, fmt.Errorf("API returned error: %s", string(body))
46      }
47
48      var chatResponse ChatResponse
49      err = json.Unmarshal(body, &chatResponse)
50      if err != nil {
51          return nil, fmt.Errorf("error parsing chat response: %v", err)
52      }
53
54      if len(chatResponse.Choices) == 0 {
55          return nil, fmt.Errorf("no choices in response")
56      }
57
58      // Очищаем ответ от thinking тегов модели
59      content := chatResponse.Choices[0].Message.Content
60      if idx := strings.Index(content, "</think>\n\n"); idx != -1 {
61          content = content[idx+11:] // +11 чтобы пропустить "</think>\n\n"
62      }
63
64      return []byte(content), nil
65  }
```

2.3.6 AI-генерация календаря задач

Реализована функция автоматической генерации расписания задач на основе активных квестов пользователя с использованием LLM API. Алгоритм работает следующим образом:

1. Сбор контекста: Система собирает информацию о всех активных квестах пользователя, включая:
   - Метаданные квестов (ID, название, описание)
   - Активные задачи с их текущими параметрами (deadline, duration,

scheduled_start, scheduled_end, task_order)

2. Формирование промпта: Создается детальный промпт, включающий:
   — Текущую дату и время
   — Информацию о всех активных квестах и задачах
   — Пользовательский запрос (например, "Сгенерируй оптимальное расписание")
   — Правила распределения задач (учет task_order, равномерное распределение, логическое дополнение существующих данных)
3. Генерация расписания: LLM генерирует JSON-структуру с расписанием для каждой задачи:
   — scheduled_start — время начала выполнения (RFC3339)
   — scheduled_end — время окончания выполнения (RFC3339)
   — deadline — дедлайн задачи (RFC3339 или null)
   — duration — продолжительность в минутах
4. Применение расписания: Сгенерированное расписание применяется к задачам пользователя и сохраняется в базу данных через метод SetOrUpdateScheduleTasks, который использует INSERT ... ON CONFLICT DO UPDATE для атомарного обновления расписания.
5. Интеграция с календарем: Расписание может быть визуализировано в календарном интерфейсе для удобного планирования и отслеживания задач.

Данная функциональность позволяет пользователям автоматически получать оптимальное расписание выполнения задач с учетом их текущей загрузки, дедлайнов и последовательности выполнения.

2.3.7 Система дружеских взаимодействий и совместных квестов

Реализована система социальных взаимодействий, включающая управление дружескими связями и создание совместных квестов.

Добавление друзей:

1. Пользователь может добавить друга по username через endpoint POST /friends/by-name/:friendName
2. Система проверяет существование пользователя с указанным username
3. Проверяется отсутсвтие в таблице friends дружбы между пользователями.

4. Создается запись в таблице friends со статусом 'accepted' (упрощенная модель без подтверждения)

Создание совместных квестов: Совместные квесты позволяют двум друзьям проходить квест вместе с синхронным завершением. Процесс реализован в методе CreateSharedQuest и выполняется в рамках транзакции:

1. Проверка дружбы: Верификация того, что пользователи являются друзьями (проверка в обоих направлениях связи)
2. Создание shared quest: Создается запись в таблице shared_quests, связывающая двух пользователей и квест
3. Покупка и старт квеста для обоих пользователей: Для каждого пользователя выполняется:
   — Проверка наличия квеста (если не куплен — покупка)
   — Проверка достаточности средств
   — Списывание монет
   — Создание записей в user_quests и user_tasks
   — Автоматический старт квеста (статус меняется на 'started')
   — Активация всех задач (статус меняется на 'active')
   — Установка времени истечения на основе time_limit_hours
4. Синхронное завершение: Квест считается завершенным только когда все задачи выполнены обоими пользователями. При завершении квеста одним пользователем система проверяет статус выполнения у второго пользователя.

Все операции выполняются атомарно в рамках одной транзакции, что гарантирует целостность данных и синхронизацию состояния квеста для обоих участников.

2.3.8 Интеграция с Recommendation Service

Основной backend взаимодействует с Recommendation Service (Python микросервис) через HTTP REST API. Реализованы следующие интеграционные точки:

Добавление квестов в индекс: При создании нового квеста (включая AI-генерацию) выполняется асинхронный запрос к Recommendation Service для добавления квеста в индекс поиска:

— Endpoint: POST /api/quests/add

— Данные: ID, название, описание, категория квеста
— Выполнение: в отдельной goroutine для неблокирующей обработки
— Хранение: данные сохраняются одновременно в SQLite (для персистентности) и в in-memory кэш (для быстрого доступа)

Добавление пользователей: При регистрации или выполнении квестов пользователь может быть добавлен в Recommendation Service для построения семантических профилей:

— Endpoint: POST /api/users/add
— Данные: user_id и список ID приобретенных квестов
— Позволяет системе строить семантические профили пользователей на основе BERT-эмбеддингов добавленных пользователем квестов
— Хранение: данные сохраняются одновременно в SQLite (для персистентности) и в in-memory кэш (для быстрого доступа)

Синхронизация с PostgreSQL: Для обеспечения консистентности данных между основным backend и Recommendation Service реализован механизм синхронизации:

— Endpoint: POST /api/sync
— Функциональность: получение актуальных данных о квестах и пользователях из PostgreSQL основного backend
— Обновление: данные синхронизируются в SQLite Recommendation Service, после чего обновляется in-memory кэш
— Использование: позволяет Recommendation Service получать актуальные данные без необходимости постоянной отправки через API endpoints

Поиск квестов:

— Endpoint: POST /api/search
— Параметры: текстовый запрос, категория, статус, top_K
— Возвращает: список квестов с оценками сходства (TF-IDF или BERT)
— Backend получает IDs квестов и загружает полные данные из PostgreSQL

Рекомендация квестов:

— Endpoint: POST /api/quests/recommend
— Параметры: список ID квестов пользователя, категория, top_K
— Алгоритм: BERT (семантический анализ приобретенных квестов пользователя)

— Возвращает: рекомендации с объяснениями и оценками сходства

Рекомендация друзей:

— Endpoint: POST /api/users/recommend
— Параметры: user_id, top_K
— Алгоритм: Collaborative Filtering на основе схожести интересов
— Backend фильтрует уже существующих друзей из результатов

Все запросы к Recommendation Service выполняются с таймаутом 30 секунд для предотвращения зависаний. Ошибки логируются, но не блокируют основную функциональность приложения.

2.3.9 Обработка голосового ввода (Web Speech API)

Для удобства пользователей реализована интеграция с Web Speech API браузера, позволяющая преобразовывать голосовой ввод в текст. Данная функциональность реализована на клиентской стороне и используется для голосового ввода промпта для генерации квестов через AI

Web Speech API обеспечивает распознавание речи в реальном времени с поддержкой различных языков и диалектов, что повышает удобство использования системы, особенно на мобильных устройствах.

2.3.10 Система транзакций и аудит финансовых операций

Все финансовые операции (покупка квестов, начисление наград) логируются в таблице user_coin_transactions для обеспечения полной прозрачности и возможности аудита. Каждая транзакция содержит:

— user_id — идентификатор пользователя
— amount — сумма транзакции (положительная для начислений, отрицательная для списаний)
— transaction_type — тип транзакции ('earned' или 'spent')
— reference_type — тип связанной сущности ('quest', 'task')
— reference_id — идентификатор связанной сущности
— description — текстовое описание транзакции
— created_at — временная метка операции

Такая система позволяет отслеживать все финансовые операции пользователя, анализировать паттерны расходования монет и обеспечивать прозрачность игровой экономики.

2.3.11 Реализация frontend-приложения

Frontend-приложение разработано на HTML, CSS и JavaScript без использования фреймворков, что обеспечивает минимальные зависимости и быструю загрузку страницы. Приложение представляет собой одностраничное веб-приложение (SPA).

Архитектура frontend:

Приложение организовано в виде модульной структуры с разделением на функциональные блоки:

- Модуль аутентификации — обработка регистрации, входа и управления JWT токенами через localStorage
- Модуль работы с квестами — отображение доступных квестов, покупка, старт, выполнение задач, завершение квестов
- Модуль поиска квестов — интеграция с Recommendation Service для семантического поиска на основе BERT
- Модуль AI-генерации — интерфейс для генерации квестов через LLM API с обработкой ответов
- Модуль друзей — управление дружескими связями, создание совместных квестов
- Модуль календаря — визуализация расписания задач с использованием FullCalendar
- Модуль карты квестов — визуализация связей между квестами с использованием Cytoscape.js

Технологический стек frontend:

- HTML5 — семантическая разметка с использованием современных элементов
- CSS3 — стилизация с использованием CSS Variables для темизации, Flexbox и Grid для адаптивной верстки
- Vanilla JavaScript (ES6+) — модульная организация кода с использованием async/await для асинхронных операций
- Fetch API — взаимодействие с REST API backend-сервиса
- FullCalendar — библиотека для отображения календаря задач
- Cytoscape.js — библиотека для визуализации графов связей между квестами
- Web Speech API — поддержка голосового ввода для поиска квестов

Взаимодействие с backend:

Все взаимодействие с backend осуществляется через единую функцию apiCall, которая:

— Автоматически добавляет JWT токен в заголовок Authorization
— Обрабатывает ошибки сети и HTTP статусы
— Предоставляет единообразный интерфейс для всех API запросов
— Реализует обработку таймаутов и повторные попытки при сбоях

Основные функциональные возможности:

1. Система табов — навигация между разделами приложения (Авторизация, Профиль, Квесты, AI Квесты, Поиск, Друзья, Мои квесты, Календарь, Карта квестов)
2. Управление состоянием — JWT токен хранится в localStorage, состояние приложения управляется через DOM манипуляции
3. Динамическое обновление контента — все данные загружаются асинхронно через Fetch API и отображаются в реальном времени
4. Обработка ошибок — система уведомлений для отображения успешных операций, ошибок и предупреждений
5. Адаптивный дизайн — интерфейс адаптируется под различные размеры экранов с использованием CSS Grid и Flexbox
6. Визуализация данных — календарь задач, граф связей квестов, карточки квестов с метаданными

Frontend интегрирован с Recommendation Service и LLM API через основной backend (детали интеграции описаны в разделе 2.3.8). Пользовательский интерфейс предоставляет возможность семантического поиска квестов и AI-генерации персонализированного контента с отображением результатов в реальном времени. Полная реализация ключевых модулей frontend-приложения представлена в приложении Ж.

2.4 Архитектура базы данных и управление транзакциями

Спроектирована реляционная база данных на PostgreSQL, отражающая все основные бизнес-сущности системы (см. приложение А). База данных включает 11 таблиц, организованных в логические группы [27]. ER-диаграмма базы данных представлена на рис. 2.

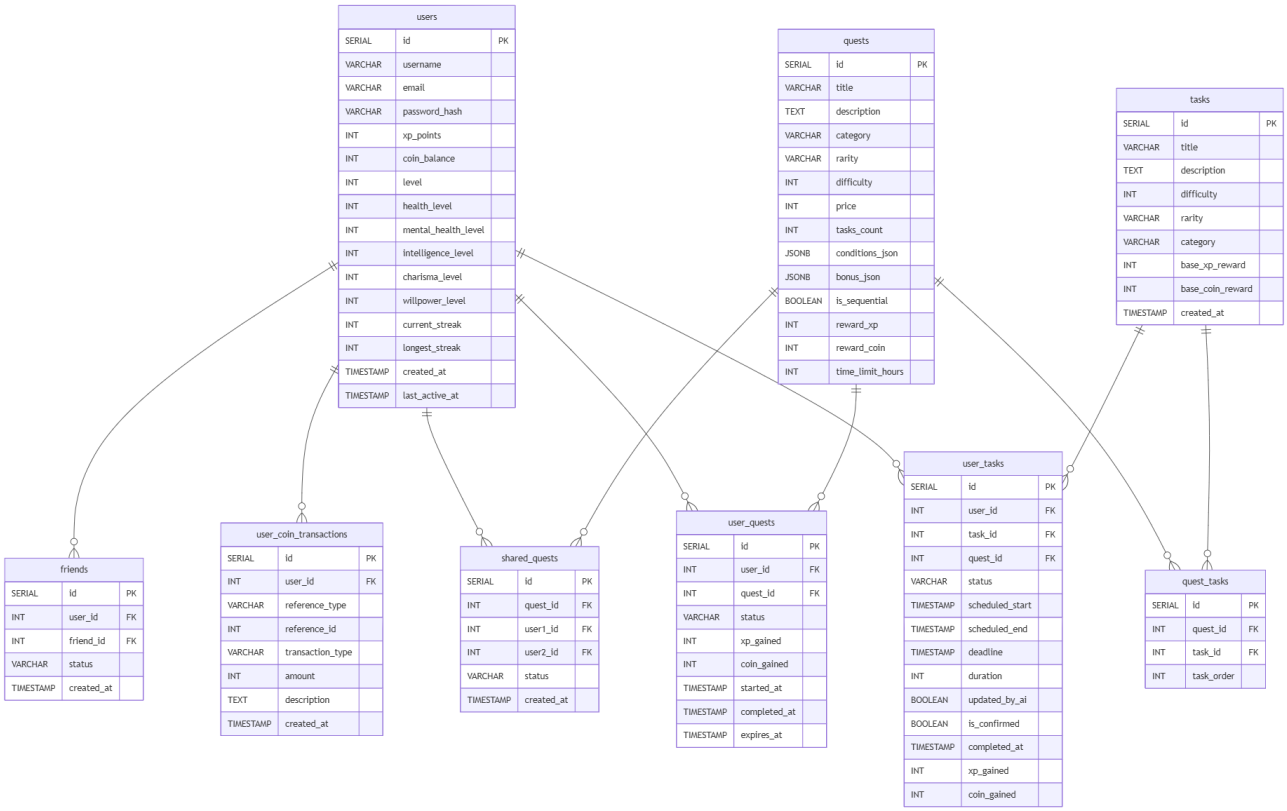

Рисунок 2 – ER-диаграмма базы данных системы

2.4.1 Основные таблицы и их назначение

Пользователи и прогресс:

— users - основная таблица пользователей с учетными данными (username, email, password_hash), статистикой прогресса (xp_points, coin_balance, level)

— user_coin_transactions - полная история всех операций с внутриигровой валютой для обеспечения прозрачности и аудита финансовых операций

Задачи и выполнение:

— tasks - каталог всех доступных задач с метаданными (title, description, difficulty, rarity, category) и системой наград (base_xp_reward, base_coin_rewar

— user_tasks - связь пользователей с задачами, содержащая статус выполнения (not_started, active, completed), расписание (scheduled_start, scheduled_end, deadline, duration), награды за выполнение (xp_gained, coin_gained) и флаг подтверждения (is_confirmed)

— quest_tasks - связующая таблица для композиции квестов из задач с указанием порядка выполнения (task_order) для поддержки последовательных квестов

Квестовая система:

— quests - метаданные квестов, включающие структурные параметры (title, description, category, rarity, difficulty, price), условия прохождения (conditions_json, is_sequential, time_limit_hours), систему наград (reward_xp, reward_coin) и опциональные бонусы (bonus_json)
— user_quests - прогресс пользователей по квестам с отслеживанием статуса (purchased, started, completed), временных меток (started_at, completed_at, expires_at) и полученных наград (xp_gained, coin_gained)

Социальные взаимодействия:

— friends - система дружеских связей с двунаправленными отношениями (проверка в обоих направлениях) и статусом связи (accepted)
— shared_quests - совместные квесты, связывающие двух пользователей (user1_id, user2_id) с квестом и отслеживающие статус совместного прохождения

2.4.2 Управление транзакциями и обеспечение целостности данных

Все критические операции в системе выполняются в рамках транзакций PostgreSQL для обеспечения ACID-свойств (Atomicity, Consistency, Isolation, Durability). Используется библиотека sqlx с методами BeginTxx для создания транзакций с поддержкой контекста. Детальное описание реализации транзакций для конкретных операций (покупка квестов, выполнение задач, создание совместных квестов) представлено в разделе "Управление квестами и задачами с использованием транзакций".

Паттерн работы с транзакциями:

1. Создание транзакции: tx, err := r.db.BeginTxx(ctx, nil)
2. Установка отката при ошибке: defer tx.Rollback()
3. Выполнение операций в рамках транзакции
4. Подтверждение: return tx.Commit() или откат при ошибке

Использование транзакций гарантирует, что система всегда находится в консистентном состоянии, даже при возникновении ошибок или сбоев.

2.4.3 Ключевые бизнес-сущности и их взаимосвязи

Пользователь (users) является центральной сущностью системы и связан со всеми остальными таблицами через внешние ключи. Содер-

жит информацию об учетных данных, прогрессе по уровням развития (с квадратичной прогрессией) и балансе валюты. Уровень пользователя автоматически пересчитывается при начислении опыта.

Задача (tasks) характеризуется метаданными (название, описание), параметрами сложности (difficulty, rarity, category) и системой наград за выполнение (base_xp_reward, base_coin_reward). Задачи могут существовать независимо или быть частью квестов.

Квест (quests) включает структурные параметры (название, описание, категория, редкость, сложность, цена), условия прохождения (is_sequential для последовательных квестов, time_limit_hours для ограничения по времени), систему наград (reward_xp, reward_coin) и опциональные условия и бонусы в формате JSONB. Квесты могут быть индивидуальными или совместными (через таблицу shared_quests).

Связь квестов и задач реализована через таблицу quest_tasks, которая позволяет:

— Создавать квесты из произвольного набора задач
— Определять порядок выполнения задач (task_order) для последовательных квестов
— Обеспечивать гибкость в композиции квестов

2.5 Процесс выполнения квеста и бизнес-логика

Для наглядности работы системы разработана диаграмма, иллюстрирующая полный цикл выполнения квеста пользователем (см. рис. 3).

Рисунок 3 – Диаграмма процесса выполнения квеста

2.5.1 Полный жизненный цикл квеста

Жизненный цикл квеста включает этапы: просмотр доступных квестов (с фильтрацией по уровню и балансу), покупка квеста (транзакционное списание монет и создание записей в user_quests и user_tasks), старт квеста (активация всех задач), выполнение задач (с немедленным начислением наград) и завершение квеста (начисление финальных наград и обновление уровня). Все операции выполняются в рамках транзакций PostgreSQL для обеспечения целостности данных. Детальное описание процесса покупки и выполнения квестов представлено в разделе "Управление квестами и задачами с использованием транзакций".

2.5.2 Особенности совместных квестов

Совместные квесты имеют уникальную бизнес-логику:

- Квест создается для двух друзей одновременно через CreateSharedQuest
- Оба пользователя должны купить квест (списываются средства с каждого)
- Квест стартуется для обоих пользователей одновременно
- Задачи выполняются независимо каждым пользователем
- Квест считается завершенным только когда все задачи выполнены обоими пользователями
- При завершении квеста одним пользователем система проверяет статус у второго
- Все участники получают полные награды за завершение квеста

Такая система обеспечивает синхронизацию прогресса и мотивирует пользователей к совместному прохождению квестов.

2.6 Реализация Recommendation Service

Recommendation Service реализован как отдельный микросервис на Python с использованием FastAPI. Сервис отвечает за все операции поиска и рекомендаций, используя различные алгоритмы машинного обучения.

Схема базы данных SQLite Recommendation Service представлена на рис. 4.

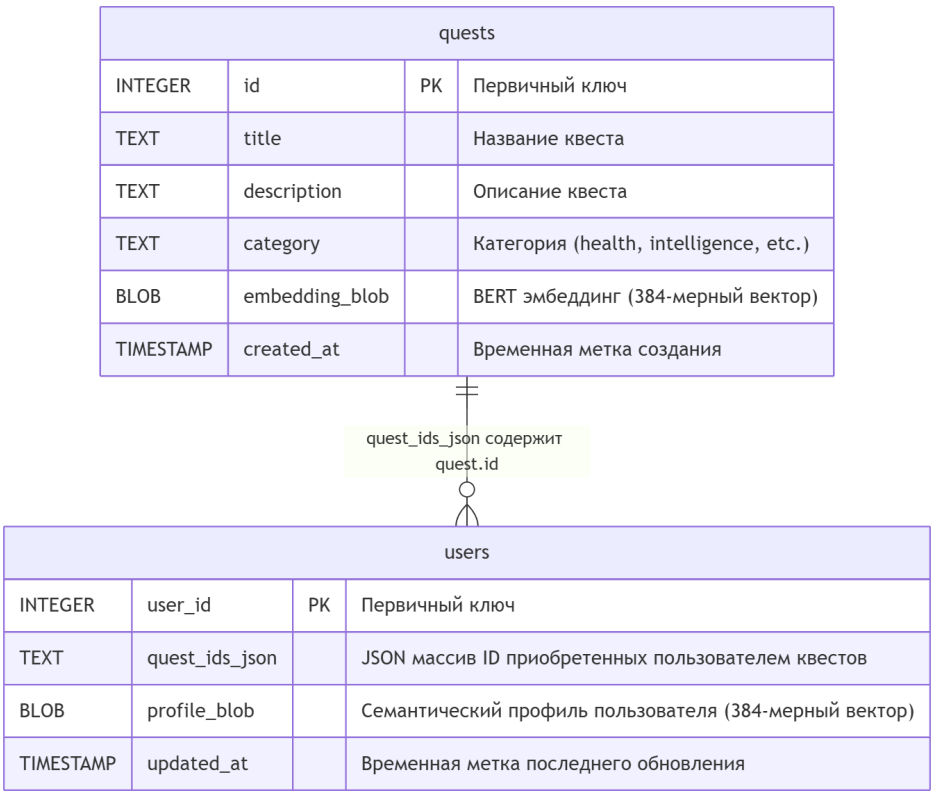

Рисунок 4 – Схема базы данных SQLite Recommendation Service

2.6.1 Архитектура Recommendation Service

Сервис реализован как FastAPI приложение с использованием архитектуры, основанной на состоянии приложения (app.state) для хранения модели и кэша данных. При инициализации сервиса (через lifespan) выполняются следующие операции:

— Загрузка модели BERT: Предобученная многоязычная модель paraphrase-multilingual-MiniLM-L12-v2 [17] загружается в память и сохраняется в app.state.model. Модель автоматически использует GPU (CUDA), если доступен, иначе CPU.

— Инициализация кэша: Создаются in-memory структуры данных для хранения эмбеддингов и метаданных:

  – app.state.quest_embeddings — словарь эмбеддингов квестов (ключ: quest_id, значение: BERT-эмбеддинг)
  – app.state.quests_data — метаданные квестов (название, описание, категория)
  – app.state.users_data — данные пользователей (список ID квестов пользователя)
  – app.state.profile_embeddings — семантические профили поль-

зователей (усредненные эмбеддинги их квестов)

— Загрузка данных из SQLite: При старте сервиса все данные загружаются из SQLite базы данных [22] в память через SQLiteBlobStorage.get_all_c и SQLiteBlobStorage.get_all_users(), что обеспечивает быстрый доступ к эмбеддингам без обращения к диску при каждом запросе.

— Двухуровневое хранение: При добавлении новых данных через API эмбеддинги сохраняются одновременно в SQLite (для персистентности) и в in-memory кэш (для быстрого доступа).

Полная реализация ключевых компонентов Recommendation Service, включая инициализацию модели, обработку запросов и работу с хранилищем, представлена в приложении E.

Детальное описание алгоритмов поиска и рекомендаций (BERT, TF-IDF, Collaborative Filtering) представлено в разделе "Реализация алгоритмов рекомендаций и поиска". Основные API endpoints описаны в разделе 2.3.8.

2.7 Взаимодействие микросервисов и архитектурные решения

Разрабатываемая система построена на микросервисной архитектуре, разделяющей ответственность между основным backend-сервисом (Go) и Recommendation Service (Python). Такое разделение обеспечивает независимое масштабирование компонентов, возможность использования различных технологий для различных задач и упрощение сопровождения системы. Детальное описание взаимодействия между сервисами представлено в разделе "Взаимодействие между микросервисами"и разделе 2.3.8.

2.7.1 Обработка ошибок и отказоустойчивость

Для обеспечения отказоустойчивости системы реализованы следующие механизмы:

Таймауты для внешних запросов: Все HTTP запросы к внешним сервисам (LLM API, Recommendation Service) выполняются с таймаутом 30 секунд для предотвращения зависаний:

```
client := &http.Client{
    Timeout: 30 * time.Second,
}
```

Асинхронная обработка некритичных операций: Операции, не требующие немедленного ответа клиенту, выполняются асинхронно в goroutines:

— Добавление квестов в Recommendation Service после AI-генерации
— Обновление индексов поиска
— Логирование аналитических данных

Обработка ошибок внешних сервисов: При ошибках взаимодействия с Recommendation Service или LLM API:

— Ошибки логируются с использованием structured logging (slog)
— Основная функциональность продолжает работать (graceful degradation)
— Пользователю возвращается понятное сообщение об ошибке
— Система не падает при недоступности внешних сервисов

Многоуровневая обработка ошибок: В системе реализована многоуровневая обработка ошибок:

— На уровне репозитория транзакции автоматически откатываются при ошибках
— На уровне сервиса бизнес-ошибки преобразуются в понятные сообщения
— На уровне обработчика HTTP ошибки возвращаются с соответствующими статус-кодами
— Используется structured logging через библиотеку slog для обеспечения читаемости логов

Health checks: Реализованы endpoint'ы для проверки состояния сервисов:

— GET /tech/ping-db — проверка доступности PostgreSQL
— GET /tech/recommendation-service/health — проверка доступности Recommendation Service

2.7.2 Слоистая архитектура backend-приложения

Backend-приложение организовано в соответствии с принципами чистой архитектуры и разделено на следующие слои:

Handler Layer (handlers/):

— Обработка HTTP запросов и ответов
— Валидация входных данных через Gin binding
— Извлечение параметров из URL и тела запроса
— Вызов методов сервисного слоя

— Формирование HTTP ответов

Service Layer (services/):

— Бизнес-логика приложения

— Координация работы репозиториев

— Интеграция с внешними сервисами (LLM API, Recommendation Service)

— Обработка сложных бизнес-правил

— Трансформация данных между слоями

Repository Layer (repositories/):

— Абстракция доступа к базе данных

— Выполнение SQL запросов

— Управление транзакциями

— Маппинг данных из БД в структуры Go

Model Layer (models/):

— Определение структур данных

— Валидация бизнес-правил

— Сериализация/десериализация JSON

Такое разделение обеспечивает:

— Тестируемость — каждый слой может тестироваться независимо

— Поддерживаемость — изменения в одном слое не влияют на другие

— Переиспользуемость — бизнес-логика не привязана к HTTP

— Масштабируемость — легко добавлять новые endpoint'ы и функциональность

Детальная архитектура backend-приложения представлена на рис. 5.

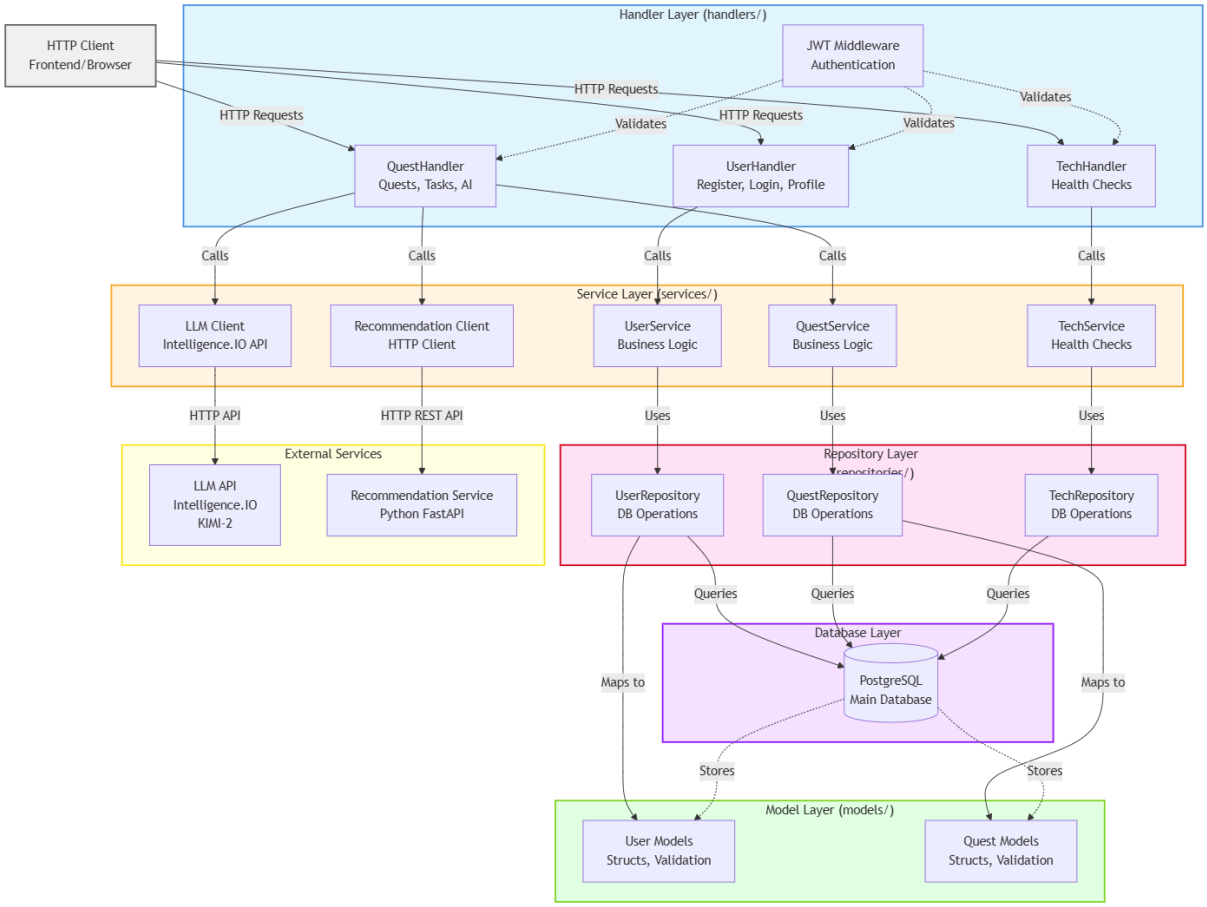

Рисунок 5 – Архитектура backend-приложения (Go)

Пример реализации слоистой архитектуры: обработчик запроса вызывает метод сервиса, который использует репозиторий для доступа к базе данных. Полная реализация ключевых компонентов представлена в приложении Д.

2.7.3 Управление конфигурацией и переменными окружения

Система использует переменные окружения для конфигурации, что обеспечивает:

— Безопасность — секретные ключи не хранятся в коде
— Гибкость — различные конфигурации для разных окружений (dev, staging, production)
— Простоту развертывания — конфигурация через environment variables

Основные конфигурационные параметры:

— JWT_SECRET — секретный ключ для подписи JWT токенов
— API_KEY_INTELLIGENCE_IO — ключ для доступа к LLM API

— DATABASE_URL — строка подключения к PostgreSQL
— RECOMMENDATION_SERVICE_BASE_URL — URL Сервиса Рекоммендаций

2.7.4 Архитектурные паттерны и принципы проектирования

При разработке системы применялись следующие архитектурные паттерны и принципы:

При разработке применялись следующие архитектурные паттерны: Repository Pattern (абстракция доступа к БД), Service Layer Pattern (централизация бизнес-логики), Dependency Injection (слабое связывание компонентов) и Transaction Script Pattern (атомарные операции в транзакциях). Эти паттерны обеспечивают независимость слоев, тестируемость, поддерживаемость и надежность системы.

2.7.5 Производительность и оптимизация

Для обеспечения высокой производительности системы применены следующие оптимизации:

Оптимизация запросов к базе данных:
— Использование SELECT только необходимых полей вместо SELECT *
— Батчинг операций при массовых вставках (например, создание user_tasks для всех задач квеста)

Оптимизация Recommendation Service:
— Двухуровневое хранение: SQLite для персистентности и in-memory кэш для быстрого доступа
— Кэширование BERT-эмбеддингов в памяти для избежания повторных вычислений
— Предвычисление BERT-эмбеддингов при добавлении квестов и сохранение их одновременно в SQLite и кэш
— Использование numpy для эффективных векторных операций с эмбеддингами
— Оптимизация вычисления косинусного сходства через векторизацию numpy для batch-обработки
— Синхронизация с PostgreSQL через HTTP endpoint для обеспечения консистентности данных

Оптимизация взаимодействия с внешними сервисами:

— Асинхронная обработка некритичных операций (добавление квестов в Recommendation Service) через goroutines
— Таймауты для всех внешних HTTP запросов (30 секунд)

Масштабируемость:

— Микросервисная архитектура позволяет независимо масштабировать Recommendation Service при росте нагрузки
— Stateless backend позволяет горизонтально масштабировать основной сервис

2.8 Тестирование и валидация системы

Для обеспечения качества разработанной системы проведено тестирование ключевых компонентов. Большинство тестов выполнялось вручную, что позволило проверить работу системы в реальных сценариях использования.

2.8.1 Тестирование алгоритмов рекомендаций

Сравнительное тестирование эффективности различных алгоритмов (TF-IDF, Collaborative Filtering, BERT) на реальных данных системы подробно описано в разделе «Реализация алгоритмов рекомендаций и поиска». Результаты показали превосходство BERT по всем метрикам качества, что привело к выбору BERT как единственного алгоритма для продакшена. Для автоматизированного тестирования API Recommendation Service использовался скрипт test_api.py (см. приложение 3), который проверяет работу всех endpoints сервиса: семантический поиск, рекомендации квестов и пользователей, поиск похожих квестов.

2.8.2 Тестирование системы цензуры KIMI 2

Проведены тесты встроенной системы цензуры модели moonshotai/Kimi-K2-Thinking:

— Тестирование на различных типах нежелательного контента
— Проверка эффективности фильтрации вредных запросов
— Результаты показали высокую эффективность системы цензуры
— Модель успешно блокирует генерацию неподходящего контента

2.8.3 Интеграционное и end-to-end тестирование

Проведено ручное тестирование взаимодействия между компонентами системы:

- Тестирование интеграции основного backend с Recommendation Service
- Валидация корректности работы транзакций при различных сценариях
- End-to-end тестирование пользовательских сценариев: регистрация, создание квестов, выполнение задач, получение рекомендаций

2.9 Направления дальнейшего развития

Разработанная система предоставляет прочную основу для дальнейшего развития. Возможные направления улучшения:

- Расширение алгоритмов рекомендаций: Интеграция более мощных моделей трансформеров (GPT, T5), fine-tuning BERT на данных системы, применение reinforcement learning для адаптации рекомендаций на основе обратной связи пользователей.
- Оптимизация производительности: Использование векторных баз данных для эффективного поиска похожих эмбеддингов, кэширование результатов рекомендаций, батчинг запросов к Recommendation Service.
- Расширение AI-функциональности: Реализация fine-tuning моделей на данных системы, добавление генерации изображений для квестов.
- Улучшение системы геймификации: Добавление системы достижений, реализация лидербордов, внедрение сезонных событий и ограниченных по времени квестов.
- Масштабирование: Использование message queues (RabbitMQ, Kafka) для асинхронной обработки, внедрение CDN для статического контента, оптимизация запросов к базе данных.
- Мониторинг и аналитика: Интеграция Prometheus и Grafana для мониторинга метрик, реализация системы аналитики пользовательского поведения, A/B тестирование алгоритмов рекомендаций.

ЗАКЛЮЧЕНИЕ

В рамках практики был спроектирован и реализован полнофункциональный адаптивный веб-приложение для саморазвития и постановки целей, включающий frontend-приложение на HTML и JavaScript, основной backend-сервис на Go, Recommendation Service на Python и интегрированную систему AI-генерации контента. Проведенная работа позволила достичь следующих результатов:

Анализ и проектирование:

1. Проведен комплексный анализ предметной области и существующих алгоритмов рекомендаций, включая Collaborative Filtering [2, 3], TF-IDF [4, 5] и современные подходы на основе трансформеров (BERT) [7, 8]. Изучены существующие системы рекомендаций (Amazon [15], Netflix [16], Spotify, YouTube) и выявлены их преимущества и ограничения.
2. Выбран и обоснован технологический стек: Go с фреймворком Gin для основного backend-разработки (высокая производительность, конкурентность), Python с FastAPI для Recommendation Service (богатая экосистема ML-библиотек), PostgreSQL в качестве основной системы хранения данных (ACID-транзакции, целостность данных), SQLite [22] для хранения эмбеддингов в Recommendation Service, HTML5 и JavaScript для frontend-разработки.
3. Спроектирована микросервисная архитектура программного комплекса с разделением на основной backend (Go) и Recommendation Service (Python). Разработана ER-диаграмма базы данных PostgreSQL, включающая 11 взаимосвязанных таблиц, а также схема базы данных SQLite для хранения BERT-эмбеддингов и семантических профилей пользователей.

Реализация backend-системы:

4. Реализована система аутентификации и авторизации на основе JWT токенов с безопасным хранением паролей через bcrypt, обеспечивающая защиту всех защищенных endpoint'ов через middleware.
5. Реализована слоистая архитектура backend-приложения (Handler → Service → Repository → Model), обеспечивающая тестируемость, поддерживаемость и масштабируемость системы.

6. Реализована система уровней игрока с квадратичной прогрессией (формула: $level = \lfloor \sqrt{XP/100} \rfloor + 1$), обеспечивающая сбалансированную прогрессию и автоматический пересчет уровня при начислении опыта.
7. Реализована система фильтрации квестов по уровню сложности, обеспечивающая постепенное открытие контента по мере прогрессии пользователя (квесты со сложностью не более чем на 1 уровень выше текущего уровня пользователя).
8. Реализована транзакционная обработка всех критических операций (покупка квестов, выполнение задач, создание совместных квестов) с использованием PostgreSQL транзакций [21, 27], гарантирующая атомарность и целостность данных.
9. Реализована система аудита финансовых операций через таблицу user_coin_transactions, обеспечивающая полную прозрачность всех операций с внутриигровой валютой.
10. Реализована система дружеских взаимодействий с поддержкой добавления друзей по username и создания совместных квестов с синхронным завершением, обеспечивающая социальную мотивацию пользователей.

Интеграция AI и машинного обучения:

13. Реализована интеграция с LLM API (Intelligence.IO Solutions, модель KIMI 2) для AI-генерации персонализированных квестов на основе текстовых запросов пользователей. Модель обладает встроенной системой цензуры, эффективно предотвращающей генерацию вредного контента (проведены тесты, подтвердившие эффективность).
14. Реализована AI-генерация календаря задач на основе активных квестов пользователя, позволяющая автоматически создавать оптимальное расписание выполнения задач с учетом дедлайнов, последовательности и текущей загрузки.
15. Проведено сравнительное тестирование алгоритмов рекомендаций квестов и друзей. Для рекомендаций квестов TF-IDF [4, 5] показал F1@5: 0.370, Collaborative Filtering — F1@5: 0.303, BERT [7, 8, 25, 26] — F1@5: 0.412. Для рекомендаций друзей BERT показал F1@3:

0.641 против 0.636 у TF-IDF и 0.631 у Collaborative Filtering. BERT продемонстрировал лучшие результаты по всем метрикам, что в сочетании с его универсальностью (один алгоритм для поиска квестов, рекомендации квестов и рекомендации друзей) привело к выбору BERT как единственного алгоритма для продакшена.

16. Реализованы алгоритмы поиска и рекомендаций на основе BERT [7, 8] для поиска квестов, рекомендации квестов и рекомендации друзей. Алгоритмы используют предобученную многоязычную модель paraphrase-multilingual-MiniLM-L12-v2 из библиотеки sentence-transformers [17] для генерации 384-мерных эмбеддингов и вычисления семантического сходства.
17. Разработан Recommendation Service на Python [12] с использованием FastAPI [11, 13], обеспечивающий RESTful API для взаимодействия с основным backend. Сервис реализует двухуровневое хранение данных: SQLite [22] для персистентности эмбеддингов и in-memory кэш для быстрого доступа, что обеспечивает высокую производительность при обработке запросов на рекомендации.
18. Обеспечена интеграция Recommendation Service с основным backend-приложением (Go), включая реализацию клиентских компонентов для взаимодействия с ML-сервисом, обработку ошибок и таймаутов (30 секунд) для обеспечения отказоустойчивости системы. Реализована асинхронная обработка некритичных операций (добавление квестов в Recommendation Service) через goroutines.

Реализация frontend-приложения:

20. Разработано frontend-приложение на HTML5 [23], CSS3 и JavaScript [24] без использования фреймворков, представляющее собой одностраничное веб-приложение (SPA) с модульной архитектурой. Приложение интегрировано с основным backend через REST API и обеспечивает полный функционал системы: управление квестами, поиск, AI-генерацию контента, социальные взаимодействия.
21. Реализованы специализированные модули frontend-приложения: модуль аутентификации с управлением JWT токенами, модуль работы с квестами, модуль поиска с интеграцией Recommendation Service, модуль AI-генерации, модуль друзей, модуль календаря за-

дач с использованием FullCalendar и модуль визуализации графов квестов с использованием Cytoscape.js.

22. Реализована обработка голосового ввода через Web Speech API для удобства пользователей, особенно на мобильных устройствах, а также адаптивный дизайн интерфейса с использованием CSS Grid и Flexbox для различных размеров экранов.

    Тестирование и отказоустойчивость:

23. Проведено тестирование системы, включающее сравнительное тестирование алгоритмов рекомендаций, тестирование системы цензуры LLM модели, интеграционное тестирование взаимодействия между компонентами системы и end-to-end тестирование пользовательских сценариев. Для автоматизированного тестирования Recommendation Service разработан скрипт test_api.py, проверяющий работу всех endpoints сервиса.
24. Обеспечена отказоустойчивость системы через таймауты для внешних запросов, асинхронную обработку некритичных операций, graceful degradation при недоступности внешних сервисов и многоуровневую обработку ошибок с использованием structured logging.

Реализованная система демонстрирует комплексный подход к созданию адаптивных приложений для саморазвития, объединяющий геймификацию, интеллектуальную персонализацию и автоматическую генерацию контента. Адаптивность системы обеспечивается двумя взаимодополняющими механизмами: системой рекомендаций на основе BERT [6–8] для персонализации контента и интеграцией с LLM API для генерации уникальных квестов на основе запросов пользователей.

Сравнительное тестирование алгоритмов рекомендаций (TF-IDF [4, 5], Collaborative Filtering [2, 3], BERT) показало превосходство BERT по всем метрикам [25, 26], что привело к его выбору как единственного алгоритма для продакшена. Использование транзакций PostgreSQL [21, 27] обеспечивает надежность и целостность данных во всех критических операциях. Микросервисная архитектура с разделением на основной backend и Recommendation Service обеспечивает независимое масштабирование компонентов системы.

Разработанное решение представляет собой полнофункциональную

платформу, готовую к использованию и дальнейшему развитию. Система может быть использована как основа для интеграции более сложных моделей машинного обучения, методов обработки естественного языка [14] и расширения функциональности в соответствии с потребностями пользователей. Практическая значимость работы подтверждается реализацией всех ключевых компонентов системы, включая frontend-интерфейс, backend-сервисы, ML-сервис и интеграцию с внешними API.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 The architecture of open source applications [Электронный ресурс]. — URL: https://aosabook.org/en/index.html (Дата обращения 18.11.2025). Загл. с экр. Яз. Англ.

2 Sarwar, B. Item-based collaborative filtering recommendation algorithms / B. Sarwar, G. Karypis, J. Konstan, J. Riedl // Proceedings of the 10th International Conference on World Wide Web. — 2001. — Pp. 285–295. — URL: https://doi.org/10.1145/371920.372071 (Дата обращения: 09.01.2026). Загл. с экр. Яз. Англ.

3 Ricci, F. Recommender Systems Handbook / F. Ricci, L. Rokach, B. Shapira. — 2nd edition. — New York: Springer, 2015.

4 Salton, G. Term-weighting approaches in automatic text retrieval / G. Salton, C. Buckley // Information Processing & Management. — 1988. — Vol. 24, no. 5. — Pp. 513–523.

5 Ramos, J. Using tf-idf to determine word relevance in document queries / J. Ramos // Proceedings of the First Instructional Conference on Machine Learning. — 2003. — URL: https://www.cs.unm.edu/~pdevineni/papers/ramos.pdf (Дата обращения: 09.01.2026). Загл. с экр. Яз. Англ.

6 Attention is all you need / A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin // Advances in Neural Information Processing Systems. — 2017. — Vol. 30. — URL: https://arxiv.org/abs/1706.03762 (Дата обращения: 09.01.2026). Загл. с экр. Яз. Англ.

7 Devlin, J. Bert: Pre-training of deep bidirectional transformers for language understanding / J. Devlin, M.-W. Chang, K. Lee, K. Toutanova // arXiv preprint arXiv:1810.04805. — 2018. — URL: https://arxiv.org/abs/1810.04805 (Дата обращения: 09.01.2026). Загл. с экр. Яз. Англ.

8 Reimers, N. Sentence-bert: Sentence embeddings using siamese bert-networks / N. Reimers, I. Gurevych // Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing. — 2019. —

URL: https://arxiv.org/abs/1908.10084 (Дата обращения: 09.01.2026). Загл. с экр. Яз. Англ.

9 Richardson, C. Microservices Patterns: With Examples in Java / C. Richardson. — Shelter Island: Manning Publications, 2018.

10 Fowler, M. Microservices [Электронный ресурс]. — URL: https://martinfowler.com/articles/microservices.html (Дата обращения: 09.01.2026). Загл. с экр. Яз. Англ.

11 Fastapi documentation [Электронный ресурс]. — URL: https://fastapi.tiangolo.com/ (Дата обращения: 09.01.2026). Загл. с экр. Яз. Англ.

12 Python documentation [Электронный ресурс]. — URL: https://docs.python.org/3/ (Дата обращения: 09.01.2026). Загл. с экр. Яз. Англ.

13 Ramírez, B. FastAPI Modern Python Web Development / B. Ramírez. — Birmingham: Packt Publishing, 2021.

14 Huyen, C. Designing Machine Learning Systems: An Iterative Process for Production-Ready Applications / C. Huyen. — Sebastopol: O'Reilly Media, 2022.

15 Linden, G. Amazon.com recommendations: Item-to-item collaborative filtering / G. Linden, B. Smith, J. York // IEEE Internet Computing. — 2003. — Vol. 7, no. 1. — Pp. 76–80.

16 Gomez-Uribe, C. A. The netflix recommender system: Algorithms, business value, and innovation / C. A. Gomez-Uribe, N. Hunt // ACM Transactions on Management Information Systems. — 2016. — Vol. 6, no. 4. — Pp. 13:1–13:19.

17 Sentence transformers documentation [Электронный ресурс]. — URL: https://www.sbert.net/ (Дата обращения: 09.01.2026). Загл. с экр. Яз. Англ.

18 Лучшие практики rest api [Электронный ресурс]. — URL: https://learn.microsoft.com/ru-ru/azure/architecture/best-practices/api-design (Дата обращения 21.11.2025). Загл. с экр. Яз. Рус.

19 Как написать производительный и безопасный backend на go [Электронный ресурс]. — URL: https://tproger.ru/articles/

kak-napisat-proizvoditelnyj-i-bezopasnyj-backend-na-go (Дата обращения: 20.11.2025) Загл. с экр. Яз рус.

20 Батчер, Go на практике / Батчер, Фарина. — ДМК Пресс, 2017.

21 Моргунов, PostgreSQL. Профессиональный SQL / Моргунов. — Москва: ДМК Пресс, 2025.

22 Sqlite documentation [Электронный ресурс]. — URL: https://www.sqlite.org/docs.html (Дата обращения: 09.01.2026). Загл. с экр. Яз. Англ.

23 Кириченко, HTML5 + CSS3. Основы современного WEB-дизайна / Кириченко, Хрусталев. — СПб.: Наука и Техника, 2018.

24 Кириченко, JavaScript для FrontEnd-разработчиков. Написание. Тестирование. Развертывание / Кириченко. — СПб.: Наука и Техника, 2020.

25 Sutriawan, S. Cosine similarity-based evidences selection for fact verification using sbert on the fever dataset / S. Sutriawan, S. Rustad, G. F. Shidik, Pujiono // Cogito. — 2024. — URL: https://cogito.unklab.ac.id/index.php/cogito/article/view/917/380 (Дата обращения: 09.01.2026). Загл. с экр. Яз. Англ.

26 Sutriawan, S. Performance evaluation of text embedding models for ambiguity classification in indonesian news corpus: A comparative study of tf-idf, word2vec, fasttext bert, and gpt / S. Sutriawan, S. Rustad, G. F. Shidik, Pujiono // ISI. — 2025. — URL: https://www.iieta.org/journals/isi/paper/10.18280/isi.300606 (Дата обращения: 09.01.2026). Загл. с экр. Яз. Англ.

27 Рогов, PostgreSQL 17 изнутри / Рогов. — Москва: ДМК Пресс, 2025.

ПРИЛОЖЕНИЕ А

Database Schema

```
DROP TABLE IF EXISTS user_achievements CASCADE;
DROP TABLE IF EXISTS achievements CASCADE;
DROP TABLE IF EXISTS user_coin_transactions CASCADE;
DROP TABLE IF EXISTS user_daily_streaks CASCADE;
DROP TABLE IF EXISTS user_completed_tasks CASCADE;
DROP TABLE IF EXISTS task_variants CASCADE;
DROP TABLE IF EXISTS tasks CASCADE;
DROP TABLE IF EXISTS user_completed_quests CASCADE;
DROP TABLE IF EXISTS user_current_quests CASCADE;
DROP TABLE IF EXISTS user_quests CASCADE;
DROP TABLE IF EXISTS quest_tasks CASCADE;
DROP TABLE IF EXISTS quests CASCADE;
DROP TABLE IF EXISTS users CASCADE;
DROP TABLE IF EXISTS categories CASCADE;

DROP TYPE IF EXISTS category_name CASCADE;
DROP TYPE IF EXISTS difficulty_level CASCADE;
DROP TYPE IF EXISTS task_type CASCADE;
DROP TYPE IF EXISTS rarity CASCADE;

CREATE TYPE category_name AS ENUM ('health', 'intelligence', 'charisma', '
    willpower');
CREATE TYPE rarity AS ENUM ('free', 'common', 'rare', 'epic', 'legendary');
CREATE TYPE task_type AS ENUM ('daily', 'weekly', 'special', '
    user_generated');

CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(100) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    xp_points INT DEFAULT 0,
    coin_balance INT DEFAULT 0,
    level INT DEFAULT 0,
    health_level INT DEFAULT 0,
    mental_health_level INT DEFAULT 0,
    intelligence_level INT DEFAULT 0,
    charisma_level INT DEFAULT 0,
    willpower_level INT DEFAULT 0,
    current_streak INT DEFAULT 0,
    longest_streak INT DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_active_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
CREATE TABLE tasks (
    id SERIAL PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    description TEXT,
    difficulty INT DEFAULT 0,
    rarity VARCHAR(255) NOT NULL DEFAULT 'free',
    category VARCHAR(255) NOT NULL,
    base_xp_reward INT NOT NULL DEFAULT 0,
    base_coin_reward INT NOT NULL DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE user_completed_tasks (
    id SERIAL PRIMARY KEY,
    user_id INT NOT NULL,
    task_id INT NOT NULL,
    is_confirmed BOOL DEFAULT FALSE NOT NULL,
    completed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    xp_gained INT NOT NULL,
    coin_gained INT NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
    FOREIGN KEY (task_id) REFERENCES tasks(id) ON DELETE CASCADE
);

CREATE TABLE user_coin_transactions (
    id SERIAL PRIMARY KEY,
    user_id INT NOT NULL,
    reference_type VARCHAR(50) NOT NULL,
    reference_id INT,
    transaction_type VARCHAR(50) NOT NULL,
    amount INT NOT NULL,
    description TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);

CREATE TABLE achievements (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    criteria_json JSONB NOT NULL,
    bonus_json JSONB,
    reward_xp INT DEFAULT 0,
    reward_coin INT DEFAULT 0,
    is_secret BOOLEAN DEFAULT FALSE
);

CREATE TABLE user_achievements (
```

```
    id SERIAL PRIMARY KEY,
    user_id INT NOT NULL,
    achievement_id INT NOT NULL,
    unlocked_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
    FOREIGN KEY (achievement_id) REFERENCES achievements(id) ON DELETE
        CASCADE,
    CONSTRAINT unique_user_achievement UNIQUE (user_id, achievement_id)
);

CREATE TABLE quests (
    id SERIAL PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    description TEXT,
    category VARCHAR(255) NOT NULL,
    rarity VARCHAR(255) NOT NULL,
    difficulty INT NOT NULL DEFAULT 0,
    price INT NOT NULL DEFAULT 0,
    tasks_count INT DEFAULT 1,
    conditions_json JSONB,
    bonus_json JSONB,
    is_sequential BOOLEAN DEFAULT FALSE,
    reward_xp INT NOT NULL,
    reward_coin INT NOT NULL,
    time_limit_hours INT DEFAULT 0
);

CREATE TABLE quest_tasks (
    id SERIAL PRIMARY KEY,
    quest_id INT NOT NULL,
    task_id INT NOT NULL,
    task_order INT,
    FOREIGN KEY (quest_id) REFERENCES quests(id) ON DELETE CASCADE,
    FOREIGN KEY (task_id) REFERENCES tasks(id) ON DELETE CASCADE
);

CREATE TABLE user_quests (
    id SERIAL PRIMARY KEY,
    user_id INT NOT NULL,
    quest_id INT NOT NULL,
    status VARCHAR(255) NOT NULL DEFAULT 'purchased',
    tasks_done INT DEFAULT 0,
    xp_gained INT,
    coin_gained INT,
    started_at TIMESTAMP,
    completed_at TIMESTAMP,
    expires_at TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
```

```
    FOREIGN KEY (quest_id) REFERENCES quests(id) ON DELETE CASCADE
);

CREATE TABLE friends (
    id SERIAL PRIMARY KEY,
    user_id INTEGER NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    friend_id INTEGER NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    status VARCHAR(50) DEFAULT 'pending',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(user_id, friend_id)
);

CREATE TABLE shared_quests (
    id SERIAL PRIMARY KEY,
    quest_id INTEGER NOT NULL REFERENCES quests(id) ON DELETE CASCADE,
    user1_id INTEGER NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    user2_id INTEGER NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    status VARCHAR(50) DEFAULT 'active',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

ПРИЛОЖЕНИЕ Б

JWT Authentication Implementation

Middleware для валидации JWT-токена validation и управления контекстом пользователя.

```
1  package middleware
2
3  import (
4      "BecomeOverMan/internal/services"
5      "errors"
6      "net/http"
7      "strings"
8
9      "github.com/gin-gonic/gin"
10 )
11
12 func JWTAuthMiddleware() gin.HandlerFunc {
13     return func(c *gin.Context) {
14         authHeader := c.GetHeader("Authorization")
15         if authHeader == "" || !strings.HasPrefix(authHeader, "Bearer ") {
16             c.AbortWithStatusJSON(http.StatusUnauthorized, gin.H{"error": "Missing or
                ↪  invalid token"})
17             return
18         }
19
20         tokenStr := strings.TrimPrefix(authHeader, "Bearer ")
21         claims, err := services.ValidateJWT(tokenStr)
22         if err != nil {
23             c.AbortWithStatusJSON(http.StatusUnauthorized, gin.H{"error": "Invalid
                ↪  token"})
24             return
25         }
26
27         c.Set("user_id", claims.UserID)
28         c.Next()
29     }
30 }
31
32 func GetUserID(c *gin.Context) (int, error) {
33     userIDKey, exists := c.Get("user_id")
34     if !exists {
```

```
35          return 0, errors.New("Cannot get user_id from context")
36      }
37
38      userID, ok := userIDKey.(int)
39      if !ok {
40          return 0, errors.New("User ID is not integer")
41      }
42
43      return userID, nil
44  }
```

JWT service для генерации и валидации токена.

```
1  package services
2
3  import (
4      "os"
5      "time"
6
7      "github.com/golang-jwt/jwt/v5"
8  )
9
10 var jwtKey = []byte(os.Getenv("JWT_SECRET"))
11
12 type Claims struct {
13     UserID int  `json:"user_id" `
14     jwt.RegisteredClaims
15 }
16
17 func GenerateJWT(userID int) (string, error) {
18     expirationTime := time.Now().Add(24 * time.Hour)
19     claims := &Claims{
20         UserID: userID,
21         RegisteredClaims: jwt.RegisteredClaims{
22             ExpiresAt: jwt.NewNumericDate(expirationTime),
23         },
24     }
25
26     token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
27     return token.SignedString(jwtKey)
28 }
29
```

```
30 func ValidateJWT(tokenStr string) (*Claims, error) {
31     claims := &Claims{}
32     token, err := jwt.ParseWithClaims(tokenStr, claims, func(token *jwt.Token)
       ↪   (interface{}, error) {
33         return jwtKey, nil
34     })
35 
36     if err != nil || !token.Valid {
37         return nil, err
38     }
39 
40     return claims, nil
41 }
```

ПРИЛОЖЕНИЕ В

AI Quest Generation Implementation

Реализация генерации квестов через LLM API с обработкой ответа и сохранением в базу данных.

```
func (h *QuestHandler) GenerateAIQuest(c *gin.Context) {
    var request RequestAI
    if err := c.ShouldBindJSON(&request); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": "Invalid request format"})
        return
    }

    // Генерация квеста через LLM API
    aiResponse, err := h.questService.GenerateAIQuest(request.Prompt)
    if err != nil {
        c.JSON(http.StatusInternalServerError,
            gin.H{"error": "Failed to generate quest: " + err.Error()})
        return
    }

    // Сохранение квеста в БД
    questID, err := h.questService.SaveQuestToDB(aiResponse.Quest, aiResponse.Tasks)
    if err != nil {
        c.JSON(http.StatusInternalServerError,
            gin.H{"error": "Failed to save quest: " + err.Error()})
        return
    }

    // Асинхронная отправка в Recommendation Service
    req := models.RecommendationService_AddQuests_Request{
        Quests: []models.RecommendationService_questToAdd{
            {
                ID:          questID,
                Title:       aiResponse.Quest.Title,
                Description: aiResponse.Quest.Description,
                Category:    aiResponse.Quest.Category,
            },
        },
    }

    go func() {
```

```
37          err := h.sendQuestToRecommendationService(req)
38          if err != nil {
39              slog.Error("Failed to send quest to recommendation service", "error", err)
40          }
41      }()
42
43      c.JSON(http.StatusOK, gin.H{
44          "message":  "Quest generated successfully",
45          "quest_id": questID,
46          "quest":    aiResponse.Quest,
47          "tasks":    aiResponse.Tasks,
48      })
49  }
```

ПРИЛОЖЕНИЕ Г

Friends Quests Implementation

Методы слоя «Репозиторий» для добавления в друзья и создания дружеских квестов.

```
1  package repositories
2
3  import (
4      "BecomeOverMan/internal/models"
5      "errors"
6
7      "github.com/jmoiron/sqlx"
8  )
9
10 func (r *UserRepository) AddFriend(userID, friendID int) error {
11     var userExists bool
12     err := r.db.Get(&userExists, `
13         SELECT EXISTS(SELECT 1 FROM users WHERE id = $1)`, friendID)
14     if err != nil {
15         return err
16     }
17     if !userExists {
18         return errors.New("user not found")
19     }
20
21     var friendshipExists bool
22     err = r.db.Get(&friendshipExists, `
23         SELECT EXISTS(SELECT 1 FROM friends WHERE user_id = $1 AND
           ↪  friend_id = $2)`,
24         userID, friendID)
25     if err != nil {
26         return err
27     }
28     if friendshipExists {
29         return errors.New("friendship already exists")
30     }
31
32     _, err = r.db.Exec(`
33         INSERT INTO friends (user_id, friend_id, status)
34         VALUES ($1, $2, 'accepted')`,
35         userID, friendID)
```

```
    return err
}

func (r *UserRepository) GetFriends(userID int) ([]models.Friend, error) {
    var friends []models.Friend
    query := `
        SELECT f.*, u.username
        FROM friends f
        JOIN users u ON f.friend_id = u.id
        WHERE f.user_id = $1 AND f.status = 'accepted'
    `
    err := r.db.Select(&friends, query, userID)
    return friends, err
}
```

Метод CreateSharedQuest использует SQL-транзакции для безопасной и надежной работы с БД.

```
func (r *QuestRepository) CreateSharedQuest(user1ID, user2ID, questID int) error {
    tx, err := r.db.Beginx()
    if err != nil {
        return err
    }
    defer tx.Rollback()

    var areFriends bool
    err = tx.Get(&areFriends, `
        SELECT EXISTS(
            SELECT 1 FROM friends
            WHERE user_id = $1 AND friend_id = $2 AND status = 'accepted'
        )`, user1ID, user2ID)
    if err != nil {
        return err
    }
    if !areFriends {
        return errors.New("users are not friends")
    }

    _, err = tx.Exec(`
        INSERT INTO shared_quests (user1_id, user2_id, quest_id, status)
        VALUES ($1, $2, $3, 'active')`,
        user1ID, user2ID, questID)
```

```
25      if err != nil {
26          return err
27      }
28
29      if err := r.startQuestForUser(tx, user1ID, questID); err != nil {
30          return err
31      }
32      if err := r.startQuestForUser(tx, user2ID, questID); err != nil {
33          return err
34      }
35
36      return tx.Commit()
37  }
38
39  func (r *QuestRepository) startQuestForUser(tx *sqlx.Tx, userID, questID int) error {
40      var alreadyPurchased bool
41      err := tx.Get(&alreadyPurchased, `
42          SELECT EXISTS(SELECT 1 FROM user_quests WHERE user_id = $1 AND
            ↪  quest_id = $2)`,
43          userID, questID)
44      if err != nil {
45          return err
46      }
47
48      if !alreadyPurchased {
49          var price int
50          err := tx.Get(&price, "SELECT price FROM quests WHERE id = $1", questID)
51          if err != nil {
52              return err
53          }
54
55          var balance int
56          err = tx.Get(&balance, "SELECT coin_balance FROM users WHERE id = $1",
            ↪  userID)
57          if err != nil {
58              return err
59          }
60
61          if balance < price {
62              return errors.New("not enough coins for shared quest")
63          }
64
```

```
65          _, err = tx.Exec(`
66              INSERT INTO user_quests (user_id, quest_id, status, tasks_done)
67              VALUES ($1, $2, 'purchased', 0)`,
68              userID, questID)
69          if err != nil {
70              return err
71          }
72
73          _, err = tx.Exec(`
74              UPDATE users SET coin_balance = coin_balance - $1 WHERE id = $2`,
75              price, userID)
76          if err != nil {
77              return err
78          }
79      }
80
81      _, err = tx.Exec(`
82          UPDATE user_quests
83          SET status = 'started', started_at = NOW()
84          WHERE user_id = $1 AND quest_id = $2`,
85          userID, questID)
86
87      return err
88  }
```

ПРИЛОЖЕНИЕ Д
Backend Implementation (Go)

Реализация ключевых компонентов backend-приложения на Go, демонстрирующая слоистую архитектуру и использование транзакций PostgreSQL.

5.1 Инициализация приложения

Точка входа приложения, демонстрирующая инициализацию базы данных, создание репозиториев, сервисов и регистрацию маршрутов:

```
1  package main
2
3  import (
4      "BecomeOverMan/internal/config"
5      "BecomeOverMan/internal/handlers"
6      "log"
7      "log/slog"
8      "BecomeOverMan/internal/repositories"
9      "BecomeOverMan/internal/services"
10     "github.com/gin-contrib/cors"
11     "github.com/gin-gonic/gin"
12     "github.com/jmoiron/sqlx"
13     _ "github.com/lib/pq"
14 )
15
16 func main() {
17     slog.SetLogLoggerLevel(slog.LevelDebug)
18
19     // Подключение к базе данных
20     db, err := sqlx.Connect("postgres", config.Cfg.DatabaseURL)
21     if err != nil {
22         log.Fatal("Failed to connect to database:", err)
23     }
24     defer db.Close()
25
26     // Создание репозиториев
27     techRepo := repositories.NewTechRepository(db)
28     userRepo := repositories.NewUserRepository(db)
29     questRepo := repositories.NewQuestRepository(db)
30
31     // Создание сервисов
```

```
    techService := services.NewTechService(techRepo)
    userService := services.NewUserService(userRepo)
    questService := services.NewQuestService(questRepo, userRepo)

    // Инициализация роутера
    r := gin.Default()
    r.Use(cors.New(cors.Config{
        AllowOrigins:     []string{"*"},
        AllowMethods:     []string{"GET", "POST", "PUT", "DELETE"},
        AllowHeaders:     []string{"Origin", "Content-Type", "Authorization"},
        AllowCredentials: true,
    }))

    // Регистрация маршрутов
    handlers.RegisterTechRoutes(r, techService)
    handlers.RegisterUserRoutes(r, userService)
    handlers.RegisterQuestRoutes(r, questService)

    if err := r.Run("0.0.0.0:8080"); err != nil {
        log.Fatal("Failed to start server:", err)
    }
}
```

5.2 Handler Layer

Пример обработчика запросов, демонстрирующий валидацию входных данных и вызов сервисного слоя:

```
func (h *QuestHandler) PurchaseQuestHandler(c *gin.Context) {
    questIDStr := c.Param("questID")
    questID, err := strconv.Atoi(questIDStr)
    if err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": "Invalid quest ID"})
        return
    }

    userID, err := middleware.GetUserID(c)
    if err != nil {
        c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }

```

```
    err = h.questService.PurchaseQuest(c.Request.Context(), userID, questID)
    if err != nil {
        c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }

    c.JSON(http.StatusOK, gin.H{"message": "Quest purchased successfully"})
}
```

5.3 Service Layer

Пример сервисного слоя, демонстрирующий бизнес-логику и интеграцию с внешними сервисами:

```
func (s *QuestService) PurchaseQuest(ctx context.Context, userID, questID int) error {
    // Вызов репозитория для покупки квеста
    err := s.questRepo.PurchaseQuest(ctx, userID, questID)
    if err != nil {
        slog.Error("Failed to purchase quest", "error", err)
        return err
    }

    // Асинхронная отправка данных в Recommendation Service
    go func() {
        questIDs, err := s.getUserQuestIDs(userID)
        if err != nil {
            slog.Error("Failed to get user quest IDs", "error", err)
            return
        }

        req := models.RecommendationService_AddUsers_Request{
            Users: []models.UserWithQuestIDS{
                {UserID: userID, QuestIDs: questIDs},
            },
        }

        _, err = s.sendUserQuestToRecommendationService(req)
        if err != nil {
            slog.Error("Failed to send user quest to recommendation service", "error", err)
        }
    }()

```

```
29      return nil
30  }
```

5.4 Repository Layer с транзакциями

Полная реализация метода покупки квеста с использованием транзакций PostgreSQL:

```
1  func (r *QuestRepository) PurchaseQuest(
2      ctx context.Context,
3      userID, questID int,
4  ) error {
5      tx, err := r.db.BeginTxx(ctx, nil)
6      if err != nil {
7          return err
8      }
9      defer tx.Rollback()
10
11     // Получаем информацию о квесте
12     var quest models.Quest
13     err = tx.GetContext(ctx, &quest,
14         "SELECT * FROM quests WHERE id = $1", questID)
15     if err != nil {
16         return err
17     }
18
19     // Проверяем баланс пользователя
20     var balance int
21     err = tx.GetContext(ctx, &balance,
22         "SELECT coin_balance FROM users WHERE id = $1", userID)
23     if err != nil {
24         return err
25     }
26
27     if balance < quest.Price {
28         return errors.New("insufficient funds")
29     }
30
31     // Создаем запись в user_quests
32     _, err = tx.ExecContext(ctx, `
33         INSERT INTO user_quests (user_id, quest_id, status)
34         VALUES ($1, $2, 'purchased')`,
```

```
35          userID, questID)
36      if err != nil {
37          return err
38      }
39
40      // Создаем user_tasks для всех задач квеста
41      _, err = tx.ExecContext(ctx, `
42          INSERT INTO user_tasks (user_id, task_id, quest_id, status)
43          SELECT $1, qt.task_id, qt.quest_id, 'not_started'
44          FROM quest_tasks qt
45          WHERE qt.quest_id = $2
46          ORDER BY qt.task_order
47      `, userID, questID)
48      if err != nil {
49          return err
50      }
51
52      // Списываем монеты
53      _, err = tx.ExecContext(ctx, `
54          UPDATE users SET coin_balance = coin_balance - $1 WHERE id = $2`,
55          quest.Price, userID)
56      if err != nil {
57          return err
58      }
59
60      // Записываем транзакцию для аудита
61      _, err = tx.ExecContext(ctx, `
62          INSERT INTO user_coin_transactions
63          (user_id, amount, transaction_type, reference_type, reference_id, description)
64          VALUES ($1, $2, 'spent', 'quest', $3, 'Purchased quest: ' || $4)`,
65          userID, -quest.Price, quest.ID, quest.Title)
66      if err != nil {
67          return err
68      }
69
70      return tx.Commit()
71  }
```

ПРИЛОЖЕНИЕ Е

Recommendation Service Implementation (Python)

Реализация ключевых компонентов Recommendation Service на Python FastAPI, демонстрирующая работу с BERT-моделью, кэшированием и двухуровневым хранением данных.

6.1 Инициализация приложения и загрузка модели

Инициализация FastAPI приложения с загрузкой BERT-модели и данных из SQLite:

```
from fastapi import FastAPI
from sentence_transformers import SentenceTransformer
from contextlib import asynccontextmanager
from internal.repo.db import SQLiteBlobStorage
import torch

storage = SQLiteBlobStorage()
NLP_MODEL = "paraphrase-multilingual-MiniLM-L12-v2"

@asynccontextmanager
async def lifespan(app: FastAPI):
    # Загрузка модели BERT
    logger.info("Загружаем модель BERT...")
    app.state.model = SentenceTransformer(NLP_MODEL)
    app.state.device = 'cuda' if torch.cuda.is_available() else 'cpu'
    app.state.model.to(app.state.device)
    logger.info(f"Модель загружена на {app.state.device}")

    # Инициализация in-memory кэша
    app.state.quest_embeddings = {}
    app.state.quests_data = {}
    app.state.users_data = {}
    app.state.profile_embeddings = {}

    # Загрузка данных из SQLite в память
    logger.info("Загружаем данные из базы данных...")
    app.state.quests_data, app.state.quest_embeddings = storage.get_all_quests()
    app.state.users_data, app.state.profile_embeddings = storage.get_all_users()
    logger.info(f"Загружено квестов: {len(app.state.quests_data)}")
    logger.info(f"Загружено пользователей: {len(app.state.users_data)}")
```

```
31
32      yield
33
34      # Закрываем БД при выключении
35      logger.info("Выключаем API...")
36      storage.close()
37
38  app = FastAPI(title="Recommendation BERT API", lifespan=lifespan)
```

6.2 Семантический поиск квестов

Реализация endpoint'а для семантического поиска квестов на основе BERT:

```
1  @app.post("/api/search")
2  async def search_quests(request: SearchRequest):
3      """Семантический поиск квестов на основе BERT"""
4      start_time = time.time()
5
6      # Генерация эмбеддинга запроса
7      query_embedding = app.state.model.encode(
8          request.query,
9          convert_to_numpy=True,
10         show_progress_bar=False
11     )
12
13     # Вычисление косинусного сходства со всеми квестами
14     similarities = []
15     for quest_id, quest_embedding in app.state.quest_embeddings.items():
16         similarity = util.cos_sim(query_embedding, quest_embedding).item()
17
18         # Фильтрация по категории, если указана
19         if request.category:
20             quest_data = app.state.quests_data.get(quest_id)
21             if quest_data and quest_data.get('category') != request.category:
22                 continue
23
24         similarities.append({
25             'quest_id': quest_id,
26             'similarity': similarity,
27             'quest_data': app.state.quests_data.get(quest_id)
28         })
```

```
29
30      # Сортировка по убыванию сходства
31      similarities.sort(key=lambda x: x['similarity'], reverse=True)
32
33      # Возврат top-K результатов
34      results = similarities[:request.top_k]
35
36      search_time_ms = (time.time() - start_time) * 1000
37
38      return {
39          'results': [
40              {
41                  'id': r['quest_id'],
42                  'title': r['quest_data']['title'],
43                  'description': r['quest_data']['description'],
44                  'category': r['quest_data'].get('category'),
45                  'similarity_score': round(r['similarity'], 4)
46              }
47              for r in results
48          ],
49          'search_time_ms': round(search_time_ms, 2)
50      }
```

6.3 Рекомендация квестов на основе профиля пользователя

Реализация алгоритма рекомендации квестов с использованием семантического профиля пользователя:

```
1  @app.post("/api/quests/recommend")
2  async def recommend_quests(request: RecommendQuestsRequest):
3      """Рекомендация квестов на основе семантического профиля пользователя"""
4
5      user_quest_ids = request.user_quest_ids
6
7      if not user_quest_ids:
8          # Для новых пользователей возвращаем популярные квесты
9          all_quests = list(app.state.quests_data.values())
10         return {
11             'recommendations': all_quests[:request.top_k],
12             'message': 'Новый пользователь: показаны популярные квесты'
13         }
14
```

```
    # Получение или вычисление семантического профиля пользователя
    if request.user_id in app.state.profile_embeddings:
        user_profile = app.state.profile_embeddings[request.user_id]
    else:
        # Вычисление профиля как среднего эмбеддинга квестов пользователя
        quest_embeddings = [
            app.state.quest_embeddings[qid]
            for qid in user_quest_ids
            if qid in app.state.quest_embeddings
        ]
        if not quest_embeddings:
            return {'recommendations': [], 'message': 'Не найдено квестов для
             ↪  профиля'}

        user_profile = np.mean(quest_embeddings, axis=0)
        app.state.profile_embeddings[request.user_id] = user_profile

    # Вычисление сходства со всеми квестами
    recommendations = []
    for quest_id, quest_embedding in app.state.quest_embeddings.items():
        # Пропускаем квесты, которые уже есть у пользователя
        if quest_id in user_quest_ids:
            continue

        # Фильтрация по категории, если указана
        if request.category:
            quest_data = app.state.quests_data.get(quest_id)
            if quest_data and quest_data.get('category') != request.category:
                continue

        similarity = util.cos_sim(user_profile, quest_embedding).item()
        recommendations.append({
            'quest_id': quest_id,
            'similarity': similarity,
            'quest_data': app.state.quests_data.get(quest_id)
        })

    # Сортировка и возврат top-K
    recommendations.sort(key=lambda x: x['similarity'], reverse=True)
    top_recommendations = recommendations[:request.top_k]

```

```
    return {
        'recommendations': [
            {
                'id': r['quest_id'],
                'title': r['quest_data']['title'],
                'description': r['quest_data']['description'],
                'category': r['quest_data'].get('category'),
                'similarity_score': round(r['similarity'], 4),
                'explanation': _get_recommendation_explanation(
                    r['quest_data'], r['similarity']
                )
            }
            for r in top_recommendations
        ]
    }
```

ПРИЛОЖЕНИЕ Ж

Frontend Implementation (JavaScript)

Реализация ключевых компонентов frontend-приложения на JavaScript, демонстрирующая модульную архитектуру и взаимодействие с backend API.

7.1 Модуль работы с API

Единая функция для всех API запросов с обработкой JWT токенов и ошибок:

```
// api.js - Модуль для работы с API
const API_BASE_URL = 'http://localhost:8080';

async function apiCall(endpoint, options = {}) {
    const token = localStorage.getItem('token');

    const config = {
        ...options,
        headers: {
            'Content-Type': 'application/json',
            ...(token && { 'Authorization': `Bearer ${token}` }),
            ...options.headers
        }
    };

    try {
        const response = await fetch(`${API_BASE_URL}${endpoint}`, config);

        if (!response.ok) {
            const error = await response.json();
            throw new Error(error.error || 'Request failed');
        }

        return await response.json();
    } catch (error) {
        console.error('API call error:', error);
        throw error;
    }
}

```

```
// Примеры использования
async function getAvailableQuests() {
    return apiCall('/quests/available', { method: 'GET' });
}

async function purchaseQuest(questID) {
    return apiCall(`/quests/${questID}/purchase`, { method: 'POST' });
}

async function searchQuests(query, category = null) {
    return apiCall('/quests/search', {
        method: 'POST',
        body: JSON.stringify({ query, category, top_k: 5 })
    });
}
```

7.2 Модуль управления квестами

Обработка отображения и управления квестами:

```
// quests.js - Модуль управления квестами
class QuestManager {
    constructor() {
        this.quests = [];
    }

    async loadAvailableQuests() {
        try {
            this.quests = await getAvailableQuests();
            this.renderQuests();
        } catch (error) {
            showNotification('Ошибка загрузки квестов', 'error');
        }
    }

    renderQuests() {
        const container = document.getElementById('quests-container');
        container.innerHTML = this.quests.map(quest => `
            <div class="quest-card">
                <h3>${quest.title}</h3>
                <p>${quest.description}</p>
                <div class="quest-meta">
```

```
                <span>Категория: ${quest.category}</span>
                <span>Цена: ${quest.price} монет</span>
                <span>Сложность: ${quest.difficulty}</span>
            </div>
            <button onclick="questManager.purchaseQuest(${quest.id})">
                Купить квест
            </button>
        </div>
    `).join('');
}

async purchaseQuest(questID) {
    try {
        await purchaseQuest(questID);
        showNotification('Квест успешно куплен!', 'success');
        await this.loadAvailableQuests();
    } catch (error) {
        showNotification(error.message, 'error');
    }
}
}

const questManager = new QuestManager();
```

7.3 Модуль семантического поиска

Интеграция с Recommendation Service для семантического поиска:

```
// search.js - Модуль семантического поиска
class SearchManager {
    constructor() {
        this.setupSearchForm();
    }

    setupSearchForm() {
        const form = document.getElementById('search-form');
        form.addEventListener('submit', async (e) => {
            e.preventDefault();
            const query = document.getElementById('search-query').value;
            const category = document.getElementById('search-category').value || null;
            await this.performSearch(query, category);
        });
```

```
15      }
16
17      async performSearch(query, category) {
18          try {
19              const results = await searchQuests(query, category);
20              this.displayResults(results);
21          } catch (error) {
22              showNotification('Ошибка поиска', 'error');
23          }
24      }
25
26      displayResults(results) {
27          const container = document.getElementById('search-results');
28          container.innerHTML = results.map(result => `
29              <div class="search-result">
30                  <h4>${result.title}</h4>
31                  <p>${result.description}</p>
32                  <div class="similarity-score">
33                      Схожесть: ${(result.similarity_score * 100).toFixed(1)}%
34                  </div>
35              </div>
36          `).join('');
37      }
38  }
39
40  const searchManager = new SearchManager();
```

ПРИЛОЖЕНИЕ З

Recommendation Service API Testing Script

Скрипт для автоматизированного тестирования всех endpoints Recommendation Service. Проверяет работу семантического поиска, рекомендаций квестов и пользователей, поиска похожих квестов, а также комплексные пользовательские сценарии.

```
# test_bert_api.py
import requests
import json
import time
import random

# Базовый URL API
BASE_URL = "http://localhost:8000"

# Расширенные квесты с более подробными описаниями
quests_data = [
    {
        "id": 1,
        "title": "Утренний дружеский марафон",
        "description": "Совместный недельный челлендж для развития силы воли и
        ↪  здоровья. Ежедневные утренние пробежки, здоровое питание и
        ↪  медитация.",
        "category": "health"
    },
    # ... остальные квесты
]

# Тестовые пользователи с разными интересами
users_data = [
    {
        "user_id": 101,
        "quest_ids": [1, 4, 6]  # Фитнес-энтузиаст
    },
    # ... остальные пользователи
]

def test_api():
    print("Расширенное тестирование BERT API\n")

```

```
    # 1. Проверяем доступность API
    try:
        health_response = requests.get(f"{BASE_URL}/api/health")
        if health_response.status_code == 200:
            print(f"API работает: {health_response.json()}")
    except:
        print("Не могу подключиться к API")
        return

    # 2. Добавляем квесты
    add_response = requests.post(
        f"{BASE_URL}/api/quests/add",
        json={"quests": quests_data}
    )
    if add_response.status_code == 200:
        result = add_response.json()
        print(f"Добавлено квестов: {result.get('added', len(quests_data))}")

    # 3. Добавляем пользователей
    add_users_response = requests.post(
        f"{BASE_URL}/api/users/add",
        json={"users": users_data}
    )
    if add_users_response.status_code == 200:
        result = add_users_response.json()
        print(f"Добавлено пользователей: {result.get('total_users', len(users_data))}")

    # 4. Тестируем семантический поиск
    test_queries = [
        ("бег и физические упражнения", None),
        ("утренние привычки и медитация", "willpower"),
        ("гитара музыка творчество", "creativity"),
    ]

    for query, category in test_queries:
        search_data = {
            "query": query,
            "top_k": 3,
            "category": category
        }
        search_response = requests.post(
```

```
        f"{BASE_URL}/api/search",
        json=search_data,
        timeout=10
    )
    if search_response.status_code == 200:
        results = search_response.json()
        print(f" Запрос: '{query}' - найдено результатов: {len(results.get('results',
        ↪  []))}")

# 5. Тестируем рекомендации квестов
recommendation_tests = [
    (101, "Фитнес-энтузиаст", None),
    (102, "Творческая личность", "creativity"),
    (108, "Новый пользователь (без истории)", None)
]

for user_id, user_type, category in recommendation_tests:
    user_quest_ids = []
    for user in users_data:
        if user["user_id"] == user_id:
            user_quest_ids = user["quest_ids"]
            break

    recommend_data = {
        "user_quest_ids": user_quest_ids,
        "top_k": 5,
        "category": category
    }
    recommend_response = requests.post(
        f"{BASE_URL}/api/quests/recommend",
        json=recommend_data,
        timeout=10
    )
    if recommend_response.status_code == 200:
        results = recommend_response.json()
        print(f"Рекомендации для {user_type}: {len(results.get('recommendations',
        ↪  []))}")

# 6. Тестируем рекомендации пользователей
for user_id in [101, 102, 103]:
    recommend_users_data = {
        "user_id": user_id,
```

```
114             "top_k": 3
115         }
116         recommend_response = requests.post(
117             f"{BASE_URL}/api/users/recommend",
118             json=recommend_users_data,
119             timeout=10
120         )
121         if recommend_response.status_code == 200:
122             results = recommend_response.json()
123             print(f"Рекомендации пользователей для ID {user_id}:
                ↪  {len(results.get('results', []))}")
124 
125     print("\nТестирование завершено!")
126 
127 if __name__ == "__main__":
128     test_api()
```

Полный код скрипта (578 строк) доступен в файле RecommendationService/src/
Скрипт включает тестирование всех endpoints Recommendation Service: добавление квестов и пользователей, семантический поиск, поиск похожих квестов, рекомендации квестов и пользователей, а также комплексные пользовательские сценарии и тесты производительности.