ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ «ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет информатики, математики и компьютерных наук

Программа подготовки бакалавров по направлению 01.03.02 Прикладная математика и информатика

Осипенко Илья Леонидович

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Разработка сервиса для подбора личного страхования для физических лиц

Рецензент
ст. преподаватель
Шадрина Е.В.

Научный руководитель:
заведущий кафедры Тинькофф
Катичев.А.Р

Заведующий кафедрой:
д. ф.-м. н., Калягин В. А.

НИЖНИЙ НОВГОРОД, 2025

Содержание

1 Введение

1.1 Актуальность темы

В постоянно быстро развивающемся мире страховой сектор находится под воздействием цифровизации и постоянного изменения потребительских ожиданий. Большая палитра страховых продуктов, которая продолжает расширяться, с одной стороны, позволяет более полно удовлетворить потребности конечного пользователя, однако с другой стороны, создает и новую проблему для потребителя - информационный шум, с которым уже не легко справиться не разбирающемуся в страховой сфере человеку, чтобы принять оптимальное решение по страховому продукту.

Согласно исследованиям McKinsey & Company[1], более 70% потребителей имеют проблемы с выбором страховых продуктов из-за комплектности по сравнению условий и недостаточного осознания специфики предложений одного или другого продукта. При этом оперативность и персонализация подачи финансовых услуг четко демонстрирует значительное повышение конверсии - до 30% в соответствии с данными Deloitte[2].

Базовые способы подбора страховых продуктов, основанные на простой стандартной фильтрации с ограниченным набором параметров абсолютно не учитывают всю разнообразность индивидуальных нюансов и обстоятельств потребителя. Как итог, получается, что потребители, часто покупают либо излишнее, либо недостаточное страховое покрытие для своего конкретного случая, что приводит, по крайней мере, к неэффективному распределению финансовых средств клиента и снижению удовлетворенности потребителя.

Развертывание технологии искусственного интеллекта и машинного обучения создает новые возможности для создания систем способных помочь неразборчивому клиенту выбрать наиболее подходящий для него страховой продукт, необходимый для достижения потребностей и целей. Но для воспроизведения подобных систем недостаточно разработать алгоритмическую составляющую по генерации рекомендаций, но и разработать масштабируемую и надежную инфраструктуру, способную обрабатывать большие объемы данных.

Исходя из выше сказанного, разработка подобной системы рекомендации страховых продуктах является актуальной научно-практической проблемой, решение которой даст возможность повысить эффективность взаимодействия потребителей страховых компаний и непосредственно страховых компаний.

1.2 Постановка задачи

С точки зрения цифровизации страховых продуктов и усложнения принятия решения конечного потребителя возникает необходимость создания системы, способной формировать рекомендации страховых продуктов на основе индивидуальных параметров клиента страховой компании.

Объект исследования - приложение для предоставления услуг по страхованию и его функциональные возможности по персонализированному подбору страховых продуктов.

Предмет исследования - методы, алгоритмы и архитектурные решения для создания масштабируемой системы рекомендаций страховых продуктов.

В работе необходимо решить следующий комплекс связанных между собой задач:

1. Проанализировать традиционные подходы к построению систем для предоставления услуг в области страхования.
2. Сформировать требования к системе, а также спроектировать и разработать архитектуру приложения.
3. Разработать механизмы деградации функциональности и репликации данных для обеспечения отказоустойчивой работы системы при отказе отдельных компонентов.
4. Перевести в контейнеризованную форму разработанного решения для упрощения процесса установки и масштабирования.

Выполнение поставленных задач предоставит возможность создать высоконагруженную, масштабируемую, отказоустойчивую систему рекомендаций страховых продуктов, работающую с увеличивающейся нагрузкой, а также с возможными изменениями в бизнес-требованиях. Практическое значение разработанной системы - это повышение конверсии, а также персонализация рекомендаций страховый продуктов для потребителей, что приведет к увеличению доходности и улучшению клиентского опыта.

1.3 Практическая значимость работы

Данная работа имеет следующую практическую значимость:

1. Улучшение взаимодействия страховой компании с клиентами. Внедрение интеллектуальной системы рекомендаций позволит уменьшить время на обзвон, в поисках лучшего страхового продукта, и повысить точность совпадения предлагаемых продуктов конкретным потребностям клиента.

2. Обеспечение оптимальности страхового портфеля клиентов. Благодаря более точному выбору полисов, клиенты получают наиболее выгодное соотношение страхового покрытия и стоимости полиса и этим снижается риск недострахования или излишнего страхования. Это позволяет увеличить уровень довольства клиентов и обеспечивает улучшение отношений с страховой компанией для дальнейшего потребления ее страховых продуктов.

3. Понижение эксплуатационных затрат страховых компаний. Они позволяют уменьшить затраты на содержание штата специалистов-консультантов и агентов, а также исключить риски человеческого фактора при составлении предложений.

4. Разработка фундамента для дальнейшего развития цифровых страховых продуктов. Разрабатываемая платформа может стать основой для внедрения инновационных страховых решений, например параметрического страхования, микрострахования, страхования по запросу (on-demand insurance), что расширяет страховой рынок и привлекает новых категорий потребителей.

Таким образом, практическая значимость работы – это разработка технологического решения, которое может помочь улучшить процесс взаимодействия страховых компаний с заинтересованными лицами, повысить эффективность бизнес-процесса выбора страховых продуктов, в результате чего должно развиваться страховое дело в целом.

2 Теоретические основы

Разработка высоконагруженной системы рекомендации страховых продуктов предполагает комплексное применение современных технологических решений и методик. В данном разделе будут изложены теоретические основы, основа проектируемой системы.

2.1 Механизмы аутентификации и авторизации

Верификационные токены

Верификационные токены это криптографические примитивы, используемыми для подтверждения легитимности действий пользователей в распределенных системах. Формально верификационный токен T можно записать как функцию вида:

$$T = f(I, K, t, \Delta t) \tag{1}$$

где I — идентификатор пользователя, K — встроенный секретный ключ системы, t — метка времени генерации, Δt — период валидности токена. В проектируемой системе верификативные токены используются для проверки электронных адресов пользователей, что дает дополнительный уровень защиты от кражи аккаунта и злоупотреблений с регистрацией.

JWT (JSON Web Token) — стандартизированный механизм (RFC 7519) безопасной передачи утверждений между участниками информационного взаимодействия в формате JSON. Структурно JWT представляет собой конкатенацию трех компонентов, закодированных в Base64URL:

$$\text{JWT} = \text{Base64URL}(H) \cdot \text{Base64URL}(P) \cdot \text{Base64URL}(S) \tag{2}$$

где H — заголовок (header), содержащий метаданные о типе токена и используемом алгоритме подписи, P — полезная нагрузка (payload), содержащая утверждения (claims), и S — подпись (signature), вычисляемая по формуле:

$$S = \text{HMAC}_{\text{SHA256}}(\text{Base64URL}(H) \cdot \text{Base64URL}(P), K) \quad (3)$$

где K — секретный ключ системы. Применение JWT в разрабатываемой системе обеспечивает реализацию stateless-аутентификации, что соответствует принципам REST и способствует горизонтальному масштабированию.

2.2 Архитектурные парадигмы

REST (Representational State Transfer) [3] — это архитектурный стиль обеспечения взаимодействия между модулями системы, позволяющий спроектировать масштабируемую систему за счет независимости общающихся модулей. К основным признакам REST'а относят:

1. Клиент-серверная архитектура с разделением ответственности.
2. Отсутствие состояния (statelessness) — каждый запрос содержит всю необходимую информацию для его обработки.
3. Кэширование ресурсов для оптимизации производительности.
4. Унифицированный интерфейс взаимодействия через стандартизированные методы HTTP.
5. Многоуровневая система, обеспечивающая масштабируемость и безопасность.

Использование REST принципов при построении архитектуры приложение позволяет получить следующие преимущества:

1. Высокая степень интероперабельности компонентов.
2. Эффективное горизонтальное масштабирование.
3. Независимость клиентской и серверной частей приложения.

Горизонтальное масштабирование — методология увеличения вычислительной мощности системы путём добавления больше вычислительных узлов, а не посредством улучшения индивидуальной производительности составляющих. Формально мы можем представить производительность системы при горизонтальном масштабировании в следующем виде:

$$P_{\text{total}} = \sum_{i=1}^{n} P_i \cdot \eta(n) \tag{4}$$

где P_{total} — общая производительность, P_i — производительность отдельного i-го узла, n — соответственно количество узлов, а $\eta(n)$ — коэффициент эффективности масштабирования, учитывающий возможные потери для координации узлов.

К положительным сторонам горизонтального масштабирование системы можно отнести:

1. Повышенную отказоустойчивость за счет распределения нагрузки.
2. Экономическую эффективность по сравнению с вертикальным масштабированием.
3. Возможность динамического адаптирования вычислительных ресурсов к текущей нагрузке.

Контейнеризация — решение на уровне операционной системы, которая позволяет провести изоляцию отдельных модулей системы и зависимостей в самодостаточные контуры. Математически сущность контейнера C можно описать следующим образом:

$$C = (A, D, R, N) \tag{5}$$

где A — приложение, D — набор зависимостей, R — ресурсные ограничения, N — сетевые настройки. В рамках разрабатываемой системы для осуществления контейнерезации будет использоваться готовое решение *Docker*, которое позволяет настроить независимые окружения посредством UI, а также упро-

щает процесс управления жизненного цикла приложения и масштабирования системы.

2.3 Механизмы обеспечения надежности и мониторинга

Деградация функциональности — свойство, при происшествии которого система продолжает корректно функционировать и предоставлять доступ, несмотря на то, что может быть недоступна часть компонентов системы [9]. С математической точки зрения такой прием D можно описать следующей функцией:

$$D = 1 - \frac{F_a}{F_t} \tag{6}$$

где F_a — количество доступных функций, а F_t — общее количество функций системы. Решение данной работы предусматривает использование деградационных механизмов, позволяющие обеспечить корректное функционирование системы даже при отказе различных компонентов системы.

Репликация данных — это процесс организации работы с данными таким образом, что хранилище разбивается на множество узлов распределенной системы. По временным характеристикам репликацию делят на два основных типа:

- *Синхронную репликацию*, при которой транзакция считается завершенной только после подтверждения записи на всех репликах:

$$T_{\text{commit}} = \max(t_1, t_2, \ldots, t_n) \tag{7}$$

  где T_{commit} — время фиксации транзакции, а t_i — время записи на i-ю реплику.

- *Асинхронную репликацию*, при которой транзакция считается завершенной после записи на первичный узел:

$$T_{\text{commit}} = t_{\text{primary}} \tag{8}$$

где t_{primary} — время записи на первичный узел.

Шардирование — методология, относящаяся к принципам горизонтального масштабирование, в рамках которого хранения данных D в базе данных разбивается на несколько непересекающихся подмножеств (шардов) S_i:

$$D = \bigcup_{i=1}^{m} S_i, \quad S_i \cap S_j = \emptyset \quad \forall i \neq j \tag{9}$$

где m — количество шардов. Благодаря шардированию можно получить распределение нагрузки между узлами базы данных, что повышает общую производительность относительно операций чтения и записи данных в базу данных за счет осуществления параллельной обработки данных для каждого шарда.

Транзакционные механизмы - особый тип операций, благодаря которому легко можно добиться целостности и согласованности данных в базе данных. Эти свойства достигаются за счет того, что транзакционные механизмы удовлетворяют принципам *ACID*:

1. *Атомарность (Atomicity)* — неделимость операций: либо все изменения применяются, либо ни одно.

2. *Согласованность (Consistency)* — переход системы из одного валидного состояния в другое.

3. *Изолированность (Isolation)* — независимость параллельно выполняемых транзакций.

4. *Долговечность (Durability)* — необратимость зафиксированных изменений.

В рамках разрабатываемого решения будут использоваться транзакционные механизмы, чтобы обеспечить согласованность пользовательских данных и истории взаимодействия клиента с рекомендованными продуктами.

SLA (Service Level Agreement) — соглашение, которое условно заключается между потребителем системы и непосредственно провайдером системы, о качестве функционирования системы. К основным метрикам, по которым производится замер качества продукта, относятся:

- Доступность системы (availability), выражаемая как:

$$A = \frac{T_{\text{up}}}{T_{\text{up}} + T_{\text{down}}} \times 100\% \tag{10}$$

  где T_{up} — время работоспособности, а T_{down} — время простоя системы.

- Время отклика (response time) — интервал между инициацией запроса и получением ответа:

$$R = t_{\text{response}} - t_{\text{request}} \tag{11}$$

- Пропускная способность (throughput) — количество обрабатываемых запросов в единицу времени:

$$\Theta = \frac{N_{\text{requests}}}{T} \tag{12}$$

  где N_{requests} — количество запросов, а T — интервал времени.

Описанные теоретические концепции и методологии позволяют заложить основу для проектирования и разработки системы рекомендации страховых продуктов, которая будет соответствовать высоким требованиям относительно нагрузки системы, отказоустойчивости и масштабируемости в услвоиях динамически изменяющихся требований к приложению и нагрузки пользователей.

3 Обзор существующих решений

В рамках анализа текущего рынка систем рекомендации страховых продуктов было проведено исследование существующих решений, представленных на российском рынке. Изучение конкурентных платформ позволило выявить основные подходы к построению подобных систем, их технологические особенности и функциональные возможности. Среди наиболее значимых решений можно выделить следующие категории:

- Системы рекомендаций крупных страховых компаний (Альфа-Страхование, Т-Страхование)
- Агрегаторы страховых продуктов (Сравни.ру, Банки.ру)
- Специализированные финтех-решения с элементами рекомендательных систем

3.1 Системы рекомендаций крупных страховых компаний

3.1.1 Альфа-Страхование

Альфа-Страхование предлагает цифровую платформу для подбора страховых продуктов, включающую мобильное приложение и веб-интерфейс.

Технологические особенности:

- Монолитная архитектура на базе Java 6 и Oracle DB 11g
- Централизованное хранилище данных без шардирования
- Ограниченные возможности горизонтального масштабирования

Функциональные возможности:

- Базовая фильтрация страховых продуктов по категориям
- Ограниченная персонализация на основе демографических данных

- Интеграция с внутренними системами компании для актуализации данных о продуктах

Платформа Альфа-Страхования характеризуется высокой стабильностью работы и широким охватом страховых продуктов собственной компании. Однако система имеет ограниченные возможности персонализации и масштабирования, что снижает эффективность рекомендаций при увеличении нагрузки.

3.1.2 Т-Страхование

Т-Страхование представляет более современное решение с элементами цифровизации и автоматизации процессов подбора страховых продуктов.

Технологические особенности:

- Гибридная архитектура с элементами микросервисного подхода
- Использование PostgreSQL с ограниченным шардированием
- Частичная контейнеризация компонентов системы

Функциональные возможности:

- Расширенный набор функций, включая калькуляторы стоимости страховых продуктов
- Средний уровень персонализации с учетом финансового положения клиента
- Частичная интеграция с партнерскими сервисами

Система Т-Страхования отличается современным интерфейсом и расширенной функциональностью по сравнению с традиционными решениями. Однако гибридная архитектура с тесным взаимодействием компонентов требует вертикального масштабирования, что приводит к увеличению стоимости инфраструктуры на 30-40% при росте нагрузки всего на 15-20%.

3.2 Агрегаторы страховых продуктов

3.2.1 Сравни.ру

Сравни.ру представляет собой агрегатор финансовых продуктов, включая страховые предложения от различных компаний.

Технологические особенности:

- Микросервисная архитектура на базе Node.js и MongoDB
- Горизонтальное масштабирование с использованием Kubernetes
- Распределенное кэширование с применением Redis

Функциональные возможности:

- Многопараметрическое сравнение страховых продуктов
- Базовая рекомендательная система на основе популярности продуктов
- Интеграция с API страховых компаний для получения актуальных данных

Сравни.ру обеспечивает высокую производительность и масштабируемость, однако система рекомендаций основана преимущественно на статистических данных о популярности продуктов, без глубокого анализа индивидуальных потребностей пользователя.

3.2.2 Банки.ру

Банки.ру — крупный финансовый маркетплейс, предлагающий, помимо банковских продуктов, широкий спектр страховых услуг. Ежемесячная аудитория платформы превышает 5 млн пользователей [?].

Технологические особенности:

- Сервис-ориентированная архитектура на базе .NET Core
- Комбинированное использование SQL Server и Elasticsearch
- Частичное применение контейнеризации с Docker

Функциональные возможности:

- Расширенная система фильтрации и сортировки страховых продуктов
- Элементы персонализации на основе истории просмотров
- Интеграция с внешними источниками данных для обогащения информации о продуктах

Банки.ру предлагает развитую систему фильтрации и сравнения страховых продуктов, однако персонализация рекомендаций ограничена анализом истории просмотров без учета комплексного профиля пользователя.

3.3 Специализированные финтех-решения

3.3.1 InsurTech-стартапы

На российском рынке появляется все больше специализированных InsurTech-стартапов, предлагающих инновационные подходы к рекомендации страховых продуктов.

Технологические особенности:

- Облачная инфраструктура с использованием AWS или Google Cloud
- Применение современных технологий машинного обучения
- API-first подход к разработке

Функциональные возможности:

- Продвинутые алгоритмы персонализации на основе машинного обучения
- Интеграция с внешними источниками данных для обогащения профиля пользователя
- Предиктивная аналитика для прогнозирования потребностей клиента

InsurTech-стартапы демонстрируют высокий уровень инноваций в области персонализации рекомендаций, однако часто сталкиваются с проблемами масштабирования и интеграции с существующими системами страховых компаний.

3.4 Технологические подходы в существующих решениях

Анализ существующих решений позволяет выделить несколько ключевых технологических подходов, применяемых в системах рекомендации страховых продуктов:

1. **Архитектурные решения**:
   - Монолитная архитектура (Альфа-Страхование) - характеризуется простотой разработки и развертывания, но ограниченной масштабируемостью и гибкостью.
   - Гибридная архитектура (Т-Страхование) - представляет собой переходный этап от монолита к микросервисам, сочетая элементы обоих подходов.
   - Микросервисная архитектура (Сравни.ру) - обеспечивает высокую масштабируемость и гибкость, но требует более сложной инфраструктуры.
2. **Подходы к персонализации**:
   - Базовая фильтрация по категориям - простейший подход, не учитывающий индивидуальные особенности пользователя.
   - Контентная фильтрация - подбор продуктов на основе соответствия характеристик продукта профилю пользователя.
   - Коллаборативная фильтрация - рекомендации на основе поведения схожих пользователей.
   - Гибридные подходы - комбинация различных методов для повышения точности рекомендаций.
3. **Технологии хранения и обработки данных**:
   - Реляционные СУБД (Oracle, PostgreSQL) - обеспечивают транзакционную целостность, но ограничены в масштабируемости.
   - NoSQL решения (MongoDB, Elasticsearch) - обеспечивают высокую масштабируемость и гибкость схемы данных.

- Гибридные хранилища - комбинация реляционных и NoSQL подходов для оптимального баланса между согласованностью и масштабируемостью.

4. **Механизмы масштабирования**:

   - Вертикальное масштабирование - увеличение мощности отдельных серверов, характерно для монолитных архитектур.
   - Горизонтальное масштабирование - добавление новых узлов в кластер, применяется в микросервисных архитектурах.
   - Контейнеризация - использование Docker и Kubernetes для управления развертыванием и масштабированием приложений.

3.5 Выявленные проблемы существующих решений

На основе анализа существующих систем рекомендации страховых продуктов можно выделить следующие ключевые проблемы:

1. **Ограниченная персонализация** - большинство систем используют базовые методы фильтрации, не учитывающие полный спектр индивидуальных характеристик пользователя.

2. **Проблемы масштабируемости** - монолитные и гибридные архитектуры, применяемые в большинстве систем, имеют ограничения по горизонтальному масштабированию, что приводит к снижению производительности при высоких нагрузках.

3. **Недостаточная отказоустойчивость** - существующие системы часто не имеют эффективных механизмов деградации функциональности и репликации данных, что приводит к каскадным отказам при сбоях отдельных компонентов.

4. **Ограниченная интеграция** - закрытые экосистемы и проприетарные API затрудняют интеграцию с внешними системами и расширение функциональ-

ности. Это ограничивает возможности по обогащению данных о пользователях и страховых продуктах, что негативно сказывается на качестве рекомендаций.

3.6 Тенденции развития систем рекомендации страховых продуктов

Анализ существующих решений и рыночных тенденций позволяет выделить следующие направления развития систем рекомендации страховых продуктов:

1. **Повышение уровня персонализации** - переход от базовой фильтрации к сложным алгоритмам машинного обучения, учитывающим широкий спектр факторов и контекстную информацию.

2. **Развитие омниканальности** - обеспечение единого пользовательского опыта и доступа к рекомендациям через различные каналы взаимодействия (веб, мобильные приложения, чат-боты, голосовые помощники).

3. **Применение современных архитектурных подходов** - переход от монолитных решений к микросервисной архитектуре, обеспечивающей гибкость, масштабируемость и отказоустойчивость.

4. **Использование продвинутых технологий анализа данных** - применение методов машинного обучения, искусственного интеллекта и больших данных для повышения точности рекомендаций и прогнозирования потребностей клиентов.

Изучение существующих решений и тенденций развития систем рекомендации страховых продуктов позволяет сформировать основу для разработки современной, высоконагруженной системы, отвечающей актуальным требованиям рынка и обеспечивающей высокий уровень персонализации и пользовательского опыта.

4 Проектирование архитектуры системы

4.1 Формирование требований к системе

Составление требований для разрабатываемой системы можно назвать одним из важнейших этапов проектирования системы, так как по четко сформированным требованиям можно произвести оценку качества и функциональную возможность реализованного продукта. Требования будут являться фундаментом для последующей разработки, а также модернизации системы для удовлетворению бизнес-требованиям.

4.1.1 Функциональные требования к системе

Функциональные требования определяют набор функций F, которыми должна обладать система, а также поведение компонент, реализующих эти функции. С математической точки зрения это может определить, как:

$$F = \{f_1, f_2, ..., f_n\} \tag{13}$$

где каждая отдельная функция f_i формулируется на основании потребностей клиентов, которые должны быть удовлетворены разрабатываемым решением, а также продиктованными бизнес-требованиями. Благодаря разбивке требований на атомарные свойства системы, мы может структурировать весь процесс разработки и гарантировать полноту реализации системы.

Механизмы аутентификации и авторизации В первую очередь стоит поговорить про требования, касающиеся безопасности приложения и персонализации доступа к системе. Для этого стоит определить следующие требования, относящиеся к информационной безопасности:

1. Регистрация пользователей с обязательной валидацией уникальности идентификаторов (имя пользователя, электронная почта) и соответствия пароля установленным критериям сложности.

2. Детектирование коллизий идентификаторов с генерацией соответствующих уведомлений пользователю при попытке создания дублирующей учетной записи.

3. Верификация электронного адреса посредством отправки токена подтверждения с ограниченным временем действия Δt.

4. Аутентификация пользователя с последующей генерацией JWT-токена доступа, содержащего идентификационные данные и временную метку валидности.

Управление профилем пользователя Для формирования персонализированных рекомендаций система должна обеспечивать сбор и хранение многомерного вектора характеристик пользователя:

1. Система должна предоставлять интерфейс для ввода и модификации многокомпонентного профиля пользователя $P = (D, S, A)$, где:

   - D — вектор демографических характеристик (пол, возраст)
   - S — вектор социально-экономических параметров (профессия, уровень дохода, семейное положение)
   - A — вектор дополнительных атрибутов (наличие детей, имущественное положение, медицинские показания, частота путешествий)

2. Система должна обеспечивать атомарное обновление компонентов профиля с сохранением истории изменений для анализа динамики предпочтений.

Генерация рекомендаций Теперь стоит поговорить про основной функционал, которым должны быть удовлетворены потребности пользователя в рамках выбора страхового продукта - рекомендация страхового полиса:

1. Система должна реализовывать механизм агрегации и анализа профиля пользователя P для генерации ранжированного множества рекомендаций $R = \{r_1, r_2, ..., r_m\}$, где каждая рекомендация r_i характеризуется коэффициентом релевантности $\alpha_i \in [0, 1]$.

2. Система должна предоставлять интерфейс для ввода дополнительных параметров запроса Q, уточняющих контекст рекомендации.

3. Система должна имплементировать механизм обратной связи, учитывающий историю взаимодействия пользователя с рекомендованными продуктами $H = \{(r_i, t_i, a_i)\}$, где t_i — временная метка взаимодействия, a_i — тип действия.

Визуализация и сравнительный анализ продуктов Осталось лишь затронуть блок, относящийся к визуальной составляющей. В данном блоке будут описаны требования, касающиеся удобства и простоты использования приложения:

1. Система должна визуализировать количественную метрику соответствия α_i для каждого рекомендованного продукта.

2. Система должна реализовывать сортировку рекомендаций по убыванию коэффициента релевантности: $\alpha_1 \geq \alpha_2 \geq ... \geq \alpha_m$.

3. Система должна обеспечивать навигацию к детализированному представлению продукта с полным набором характеристик.

4. Система должна предоставлять функциональность сравнительного анализа выбранного подмножества продуктов по ключевым параметрам.

4.1.2 Нефункциональные требования к системе

Нефункциональные требования - это ограничения, налагаемые на систему, которые определяют ее атрибуты качества. Они обычно обозначаются такими терминами, как безопасность, производительность и масштабируемость. Нефункциональные требования важны, потому что они помогают гарантировать, что система соответствует потребностям пользователя.

Масштабируемость Данный тип требований описывает способность системы справляться с увеличением нагрузки без ухудшения производительности:

1. Горизонтальная масштабируемость — архитектура системы должна поддерживать линейное увеличение производительности $P(n) = k \cdot n + b$ при добавлении n вычислительных узлов, где k — коэффициент эффективности масштабирования, b — базовая производительность.

2. Контейнеризация компонентов — каждый функциональный модуль системы должен быть инкапсулирован в изолированную среду исполнения с использованием технологии Docker, что обеспечивает унификацию развертывания и управления зависимостями.

3. Автоматическое масштабирование — система должна интегрироваться с платформой оркестрации Kubernetes для динамического управления ресурсами в соответствии с функцией нагрузки $L(t)$.

Отказоустойчивость и надежность В свою очередь требования про отказоустойчивость системы и в целом про ее надежность говорят про способность системы функционировать корректно и без сбоев в течение определенного времени:

1. Дизайн архитектуры должен включать приоритетность модулями, критически важными для работы системы.

2. При сбое системы должны быть обеспечены средства защиты от каскадного отказа системы путем автоматической перезагрузки модуля с переповторами в 30 секунд.

3. При возникновении сбоев на сетевой инфраструктуре, система должна сохранить свою работоспособность, предотвращая полный отказ всех модулей системы.

Согласованность данных Для сохранения целостности информационной модели система должна внедрить:

1. Транзакционная целостность — все модификации данных должны выполняться в рамках атомарных транзакций, удовлетворяющих ACID-свойствам.

2. Компенсирующие транзакции — система должна реализовывать механизм отката изменений T^{-1} для каждой транзакции T в случае детектирования аномалий или сбоев.

Производительность Нефункциональные требования по производительности системы, описывают скорость и производительность системы, время отклика, пропускную способность.

1. Время отклика серверной части — 98-й процентиль времени обработки запросов не должен превышать 100 мс: $P_{98}(T_{response}) \leq 100$ мс.

2. Стабильность клиентской части — коэффициент доступности фронтенд-компонентов должен составлять не менее 99%: $A_{frontend} \geq 0.99$.

3. Общая доступность системы — время простоя не должно превышать 43.8 минут в месяц, что соответствует SLA 99.9%: $A_{system} \geq 0.999$.

4. Асинхронная обработка — длительные операции должны выполняться в неблокирующем режиме с использованием механизмов очередей и отложенных вычислений.

Безопасность Необходимо обеспечить безопасность проектируемой системы, более того, для защиты данных необходимо принять меры по защите и предотвращению доступа третьей стороны без разрешения. Для этого система должна отвечать следующим условиям:

1. Защита от инъекций — все пользовательские входные данные должны проходить параметризованную валидацию и санитизацию для предотвращения SQL-инъекций и других векторов атак.

2. Криптографическая защита — конфиденциальные данные должны шифроваться в состоянии покоя и при передаче с использованием современных криптографических алгоритмов.

3. Ролевая модель доступа — система должна имплементировать многоуровневую модель авторизации на основе ролей и разрешений, обеспечивающую принцип минимальных привилегий.

4.2 Пользовательские сценарии

Составление сценариев использования приложения конечными потребителями является особым инструментом, который позволяет провести малую валидацию сформулированных функциональных требований к системе на раннем этапе проектирования. Каждый расписанный сценарий в отдельности представляет полноценное взаимодействие пользователя с определенными блоками системы. Все сценарии в совокупности составляют комплексную картину того, что должно позволить делать приложение пользователю в конечном итоге. Благодаря пользовательским сценариям также можно оценить возможные проблемы взаимодействия и оптимизации пользовательского опыта, чтобы ему приходилось выполнять меньшее количество действия для получения конечного результата.

4.2.1 Регистрация и верификация электронного адреса

В рамках сценария регистрации и подтверждения электронного адреса будет расписан основной флоу, который пройдет пользователь для того, чтобы открыть полный доступ к основному функционалу системы.

Акторы: Неаутентифицированный пользователь (U), Система (S)
Предусловия: Пользователь не имеет активной учетной записи в системе
Основной поток:

1. $U \to S$: Передача регистрационных данных
   $D = (username, email, password)$
2. S: Валидация данных $V(D) \to \{true, false\}$
3. S: Создание учетной записи и профиля $A = CreateAccount(D)$

4. $S \to U$: Генерация и отправка верификационного токена
   $T = GenerateToken(email, \Delta t)$

5. $U \to S$: Активация токена $ActivateToken(T)$

6. S: Верификация токена и активация аккаунта
   $VerifyToken(T) \to SetAccountStatus(A, "active")$

Альтернативные потоки:

- Коллизия идентификаторов (шаг 2):
  $V(D) = false \to S \to U : ErrorMessage("Emailalreadyinuse")$
- Недействительный токен (шаг 5): $VerifyToken(T) = false \to$
  $\to S \to U : ErrorMessage("Invalidverificationtoken")$

4.2.2 Аутентификация и модификация профиля

Далее после успешной регистрации и соответственно подтверждение электронного адреса, пользователю открывается главный экран для ввода данных аккаунта для прохождения непосредственно авторизации. После авторизации в системе пользователю открывается возможность использовать остаточный функционал системы, однако для этого ему необходимо заполнить информацию о себе.

Акторы: Зарегистрированный пользователь (U), Система (S)
Предусловия: Пользователь имеет верифицированную учетную запись
Основной поток:

1. $U \to S$: Передача аутентификационных данных
   $D_{auth} = (email, password)$

2. S: Верификация учетных данных $Authenticate(D_{auth}) \to \{true, false\}$

3. $S \to U$: Генерация и передача токена доступа
   $JWT = GenerateJWT(user\_id, \Delta t_{session})$

4. $U \rightarrow S$: Запрос на доступ к профилю $AccessProfile(JWT)$

5. $U \rightarrow S$: Модификация профиля $UpdateProfile(JWT, P_{new})$

6. S: Валидация и сохранение изменений
   $ValidateAndSave(P_{new}) \rightarrow \{success, error\}$

Альтернативные потоки:

- Ошибка аутентификации (шаг 2): $Authenticate(D_{auth}) = false \rightarrow S \rightarrow U : ErrorMessage(\text{"}Invalidcredentials\text{"})$
- Не верифицированный аккаунт (шаг 2): $AccountStatus \neq \text{"}active\text{"} \rightarrow S \rightarrow U : ErrorMessage(\text{"}Emailnotverified\text{"})$

4.2.3 Генерация и получение рекомендаций

Следующим этапом, после того, как пользователь заполнил необходимую информацию о себе для генерации рекомендации, он может перейти на страницу с рекомендациями и нажать на кнопку для получения рекомендации.

Акторы: Аутентифицированный пользователь (U), Система (S)
Предусловия: Пользователь аутентифицирован и имеет заполненный профиль
Основной поток:

1. $U \rightarrow S$: Запрос на генерацию рекомендаций $RequestRecommendations(JWT)$

2. S: Извлечение профиля пользователя $P = GetProfile(JWT)$

3. $U \rightarrow S$: Опциональная модификация параметров запроса
   $Q = ModifyParameters(P)$

4. S: Анализ профиля и генерация рекомендаций
   $R = GenerateRecommendations(P, Q)$

5. $S \rightarrow U$: Передача ранжированного списка рекомендаций $SendRecommendations(R)$

Альтернативные потоки:

- Неполный профиль (шаг 2): $Completeness(P) < threshold \rightarrow S \rightarrow U$: $RequestAdditionalData(MissingFields(P))$
- Отсутствие релевантных рекомендаций (шаг 4): $|R| = 0 \rightarrow S \rightarrow U$: $SuggestParameterModification()$

4.2.4 Детализированный просмотр страхового продукта

И последний этап взаимодействия клиента с разрабатываемой системой - просмотр карточек рекомендованных продуктов. После получения рекомендации пользователю отображается список страховых продуктов, где о каждом есть возможность просмотреть дополнительную информацию.

Акторы: Аутентифицированный пользователь (U), Система (S)
Предусловия: Пользователь получил список рекомендаций
Основной поток:

1. U: Просмотр ранжированного списка рекомендаций $R = \{r_1, r_2, ..., r_m\}$
2. $U \rightarrow S$: Выбор конкретного продукта для детализации $SelectProduct(r_i)$
3. $S \rightarrow U$: Предоставление детализированной информации $ProductDetails(r_i) = \{description, cost, coverage, terms\}$
4. S: Регистрация взаимодействия $LogInteraction(user\_id, r_i, "view\,timestamp)$

Альтернативные потоки:

- Неполная информация о продукте (шаг 3):
  $Completeness(ProductDetails(r_i)) < 1 \rightarrow S \rightarrow U : PartialDetails(r_i) + NotificationIncomplete()$

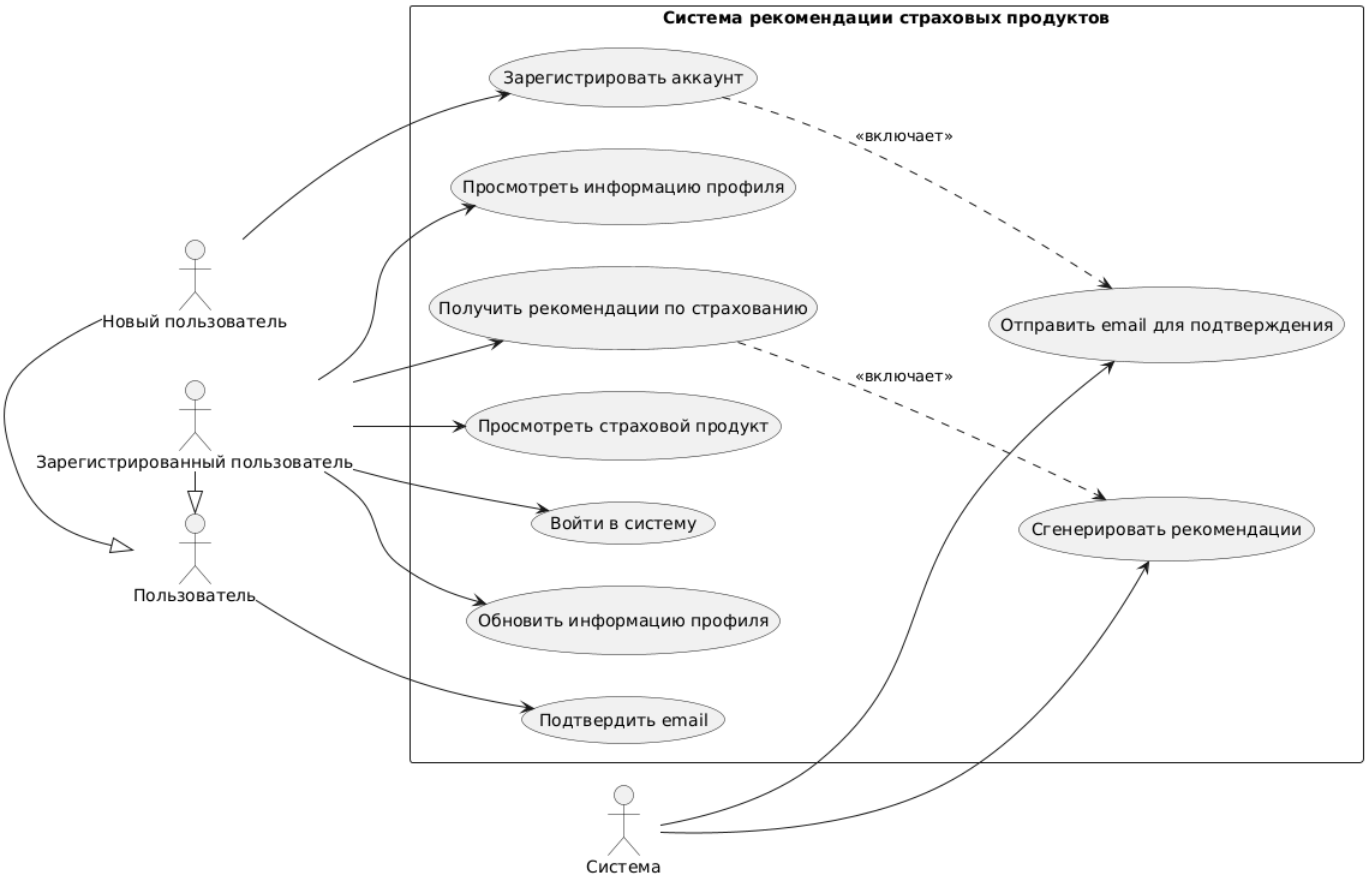

Рис. 1: Диаграмма вариантов использования системы рекомендации страховых продуктов

По сформулированным сценариям использования приложения для получения рекомендации страховых продуктов была составлена обобщенная диаграмма использования (use case) в нотации *UML*, визуализирующая основные возможности приложения с точки зрения взаимодействия с потребителем, описанные в сценариях(рис. 1).

4.3 Формирование архитектурной модели

Так как ранее были сформулированы различные требования к системе, включая функциональные и нефункциональные требования, а также составлены сцена-

рии использования необходимо приступить к разработке дизайна архитектуры системы. Удобным способом представления дизайна архитектуры будет являться отображения в виде диаграмм. Для этого отлично подойдут диаграммы нотации *C4*[1]. Непосредственно в данном разделе мы рассмотрим две ключевые диаграммы необходимые для понимания контекста приложения и того, из каких составляющих будет складываться общая архитектура системы.

4.3.1 Контекстная диаграмма

Начнем с первого и самого верхнего уровня абстракции - уровень контекста приложения. На данном уровне система описывается как единое целое и лишь отображается взаимодейтсвие разрабатываемой системы с внешними взаимосвязанными системами и потребителями. Математически это можно представить в виде ориентированного графа $G = (V, E)$, где $V = \{S\} \cup E$ — множество вершин, включающее проектируемую систему S и множество внешних систем E, а $E \subseteq V \times V \times I$ — множество ребер, представляющих информационные потоки I между вершинами.
Контекстная диаграмма обеспечивает:

1. Определение границ системы, определяя зону ответственности каждого отдельного разрабатываемого модуля.

2. Идентификацию внешних интерфейсов, через которые система взаимодействует с внешним миром.

3. Визуализацию информационных потоков между проектируемой системой и внешними сущностями.

[1]Нотация C4 - иерархический подход к визуализации архитектуры программного обеспечения. Они позволяют описывать систему на разных уровнях детализации, от общего контекста до конкретных элементов кода.

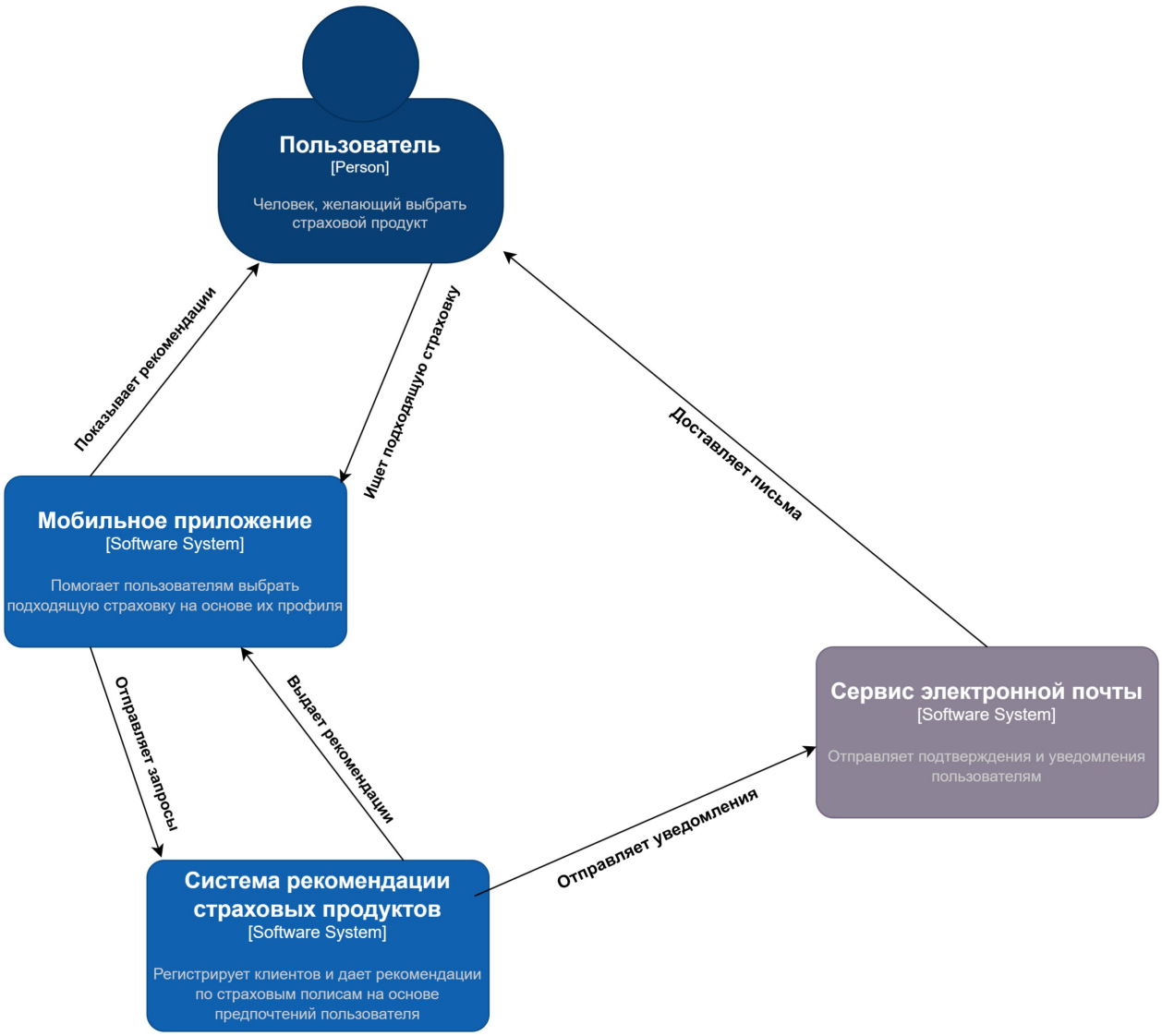

Рис. 2: Диаграмма архитектуры на уровне контекста

4.3.2 Компонентная диаграмма

Теперь поговорим про предпоследний уровень иерархии в нотации *C4* - уровень компонент. На данном уровне уже практически не отображаются внешние системы и потребители, а все внимание сосредоточено лишь на отдельных компонентах разрабатываемой системы. Смысл данной диаграммы состоит в том, что система разбивается на отдельные функциональные компоненты и указывается какие компоненты и каким образом взаимодействуют друг с другом. Аналогично диаграмме контекстной диаграмме эту можно представить в ви-

де ориентированного графа $G_c = (V_c, E_c)$, где $V_c = \{c_1, c_2, ..., c_n\}$ — множество компонентов системы, а $E_c \subseteq V_c \times V_c \times I_c$ — множество ребер, представляющих интерфейсы взаимодействия I_c между компонентами.
Компонентная диаграмма обеспечивает:

1. Декомпозицию системы на функциональные модули с четко определенными зонами ответственности.

2. Спецификацию интерфейсов взаимодействия между компонентами, определяющих контракты и протоколы обмена данными.

3. Визуализацию архитектурных паттернов и стилей, применяемых в системе.

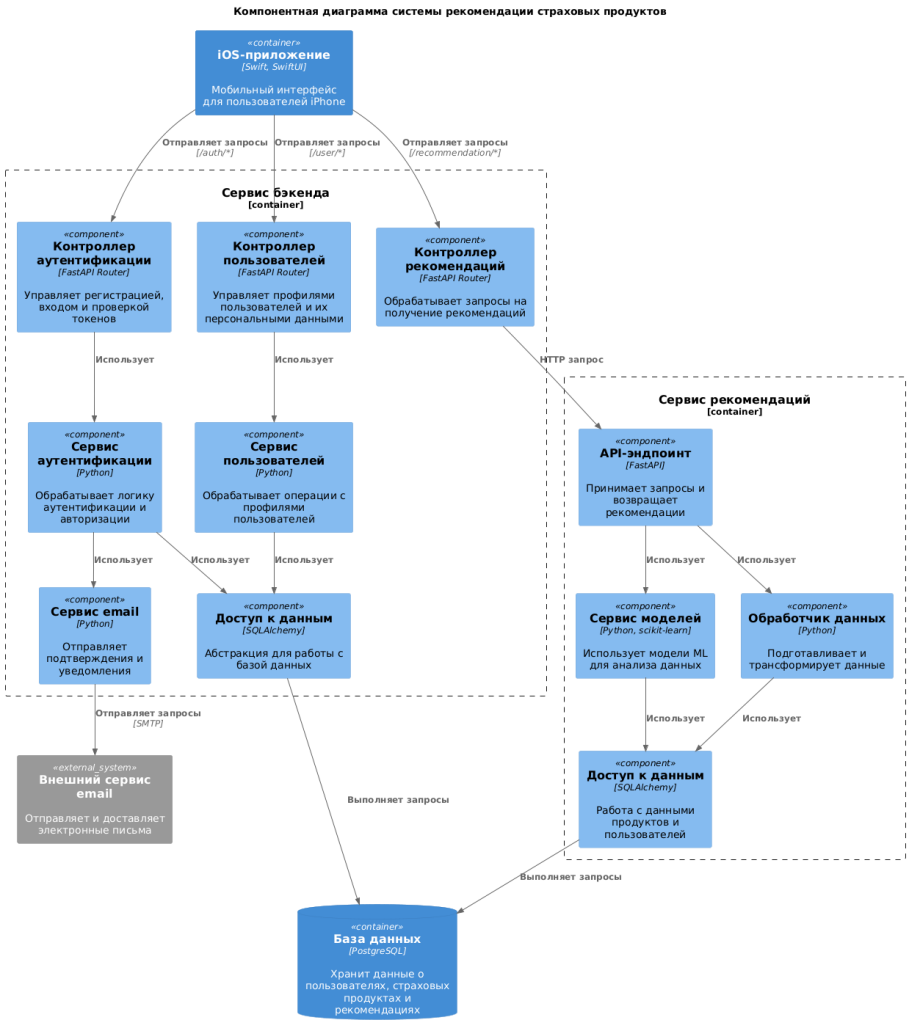

Рис. 3: Диаграмма архитектуры на уровне компонентов

Как итог, получаем две диаграммы подробно расписывающих дизайн архитектуры разрабатываемй системы. Представленные диаграммы позволяют сложить общее понимание того, как и с кем из внешних систем наша система взаимодействует. А также углубляет понимание того, из каких модулей непосредственно состоит наша система. По этим диаграммам уже можно провести валидацию проектируемой архитектуры на соответствие ранее описанным требованиям. Более того, изучая детальнее компонентную диаграмму, можно сделать вывод, что система имеет паттерн модульного монолита, в котором каждый отдельный модуль инкапсулирует свои собственные функции, необходимые в приложении, однако обращаются в общую базу данных. На рис. 4 представлен пример архитектуры модульного монолита:

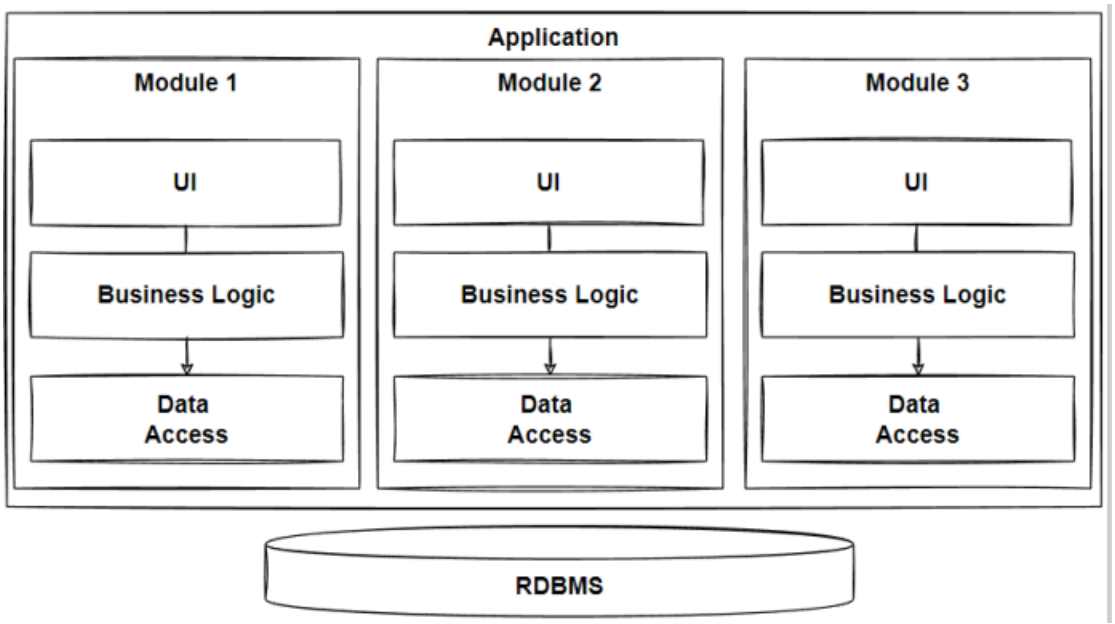

Рис. 4: Пример архитектуры модульного монолита

4.4 Проектирование базы данных

4.4.1 Выбор СУБД и обоснование

Первым делом, когда начнется разработка системы, будет реализация хранилища данных, удовлетворяющего составленным требованиям. Для того, чтобы

хранить необходимую информацию по клиентам и страховым продуктам необходимо проверить, что СУБД [2] удовлетворяла следующим условиям:

1. Поддержка горизонтального масштабирования через механизмы репликации и шардирования.
2. Обеспечение транзакционной целостности данных в соответствии с ACID-принципами.
3. Производительность операций чтения/записи при высоких нагрузках.
4. Гибкость схемы данных для адаптации к изменяющимся требованиям.

На основании проведенного анализа для реализации хранилища данных выбрана реляционная СУБД *PostgreSQL*, обеспечивающая оптимальный баланс между требуемыми характеристиками и сложностью администрирования.

PostgreSQL предоставляет следующие преимущества:

- Полная поддержка *ACID*-транзакций, критичных для обеспечения целостности данных при высоконагруженных операциях.
- Легкая настройка механизмов репликации данных из одной базы в другую. Поддерживает как синхронную репликацию, так и ассинхронную.
- Возможность горизонтального масштабирования через инструменты партиционирования и шардирования.
- Расширенные возможности индексирования, включая частичные и многоколоночные индексы, функциональные индексы и поддержку полнотекстового поиска.
- Поддержка *JSON/JSONB* форматов данных, что обеспечивает гибкость в хранении неструктурированных данных профилей пользователей и параметров страховых продуктов.

[2] СУБД - система управления базами данных

4.4.2 Логическая модель данных

Для того, чтобы предоставить общее понимание планируемого хранилища данных необходимо также составить две диаграммы - обобщенную логическую диаграмму, представленная на рис.5, в рамках которой будут продемонстрированы типы хранимых данных. А также более прикладную диаграмму на физическом уровне, где уже все атрибуты будут более подробно расписаны.
Ключевыми сущностями в проектируемой базе будут:

1. **User** — сущность, представляющая зарегистрированного пользователя системы с базовыми учетными данными для аутентификации и авторизации.
2. **UserProfile** — расширенная информация о пользователе, содержащая демографические и социально-экономические характеристики.
3. **InsuranceCategory** — категория страхового продукта (медицинское страхование, страхование жизни, автострахование и т.д.).
4. **InsuranceProduct** — базовая сущность страхового продукта с общими атрибутами, такими как стоимость, покрытие и срок действия.
5. **ProductViewHistory** — история просмотров страховых продуктов пользователями для анализа их поведения и предпочтений.

Между сущностями БД можем определить следующие типы отношений атрибутов:

- Каждый пользователь (User) имеет один профиль (UserProfile) — **один к одному**.
- Каждый страховой продукт (InsuranceProduct) принадлежит к одной категории (InsuranceCategory) — **многие к одному**.
- Пользователь (User) может иметь множество записей истории просмотров (ProductViewHistory) различных страховых продуктов — **один ко многим**.

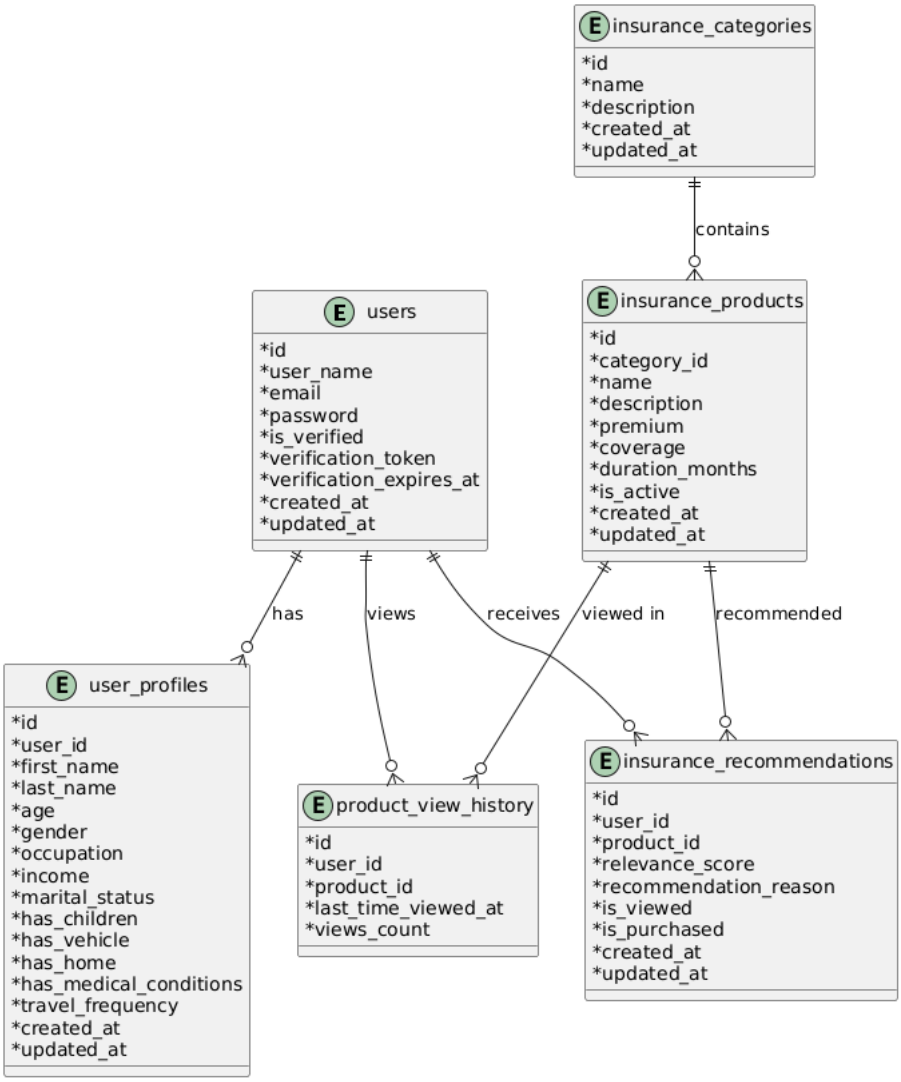

Рис. 5: ERD диаграмма модели данных на логическом уровне

Представленная логическая модель обеспечивает необходимую гибкость для последующей физической реализации и учитывает специфику предметной области страховых продуктов и персонализированных рекомендаций.

4.4.3 Физическая модель данных

На основе логической модели разработана и реализована физическая модель данных, определяющая конкретные структуры таблиц, связи и методы доступа к данным. Для модели данных физического уровня, представленная на рис.6, описаны следующие атрибуты:

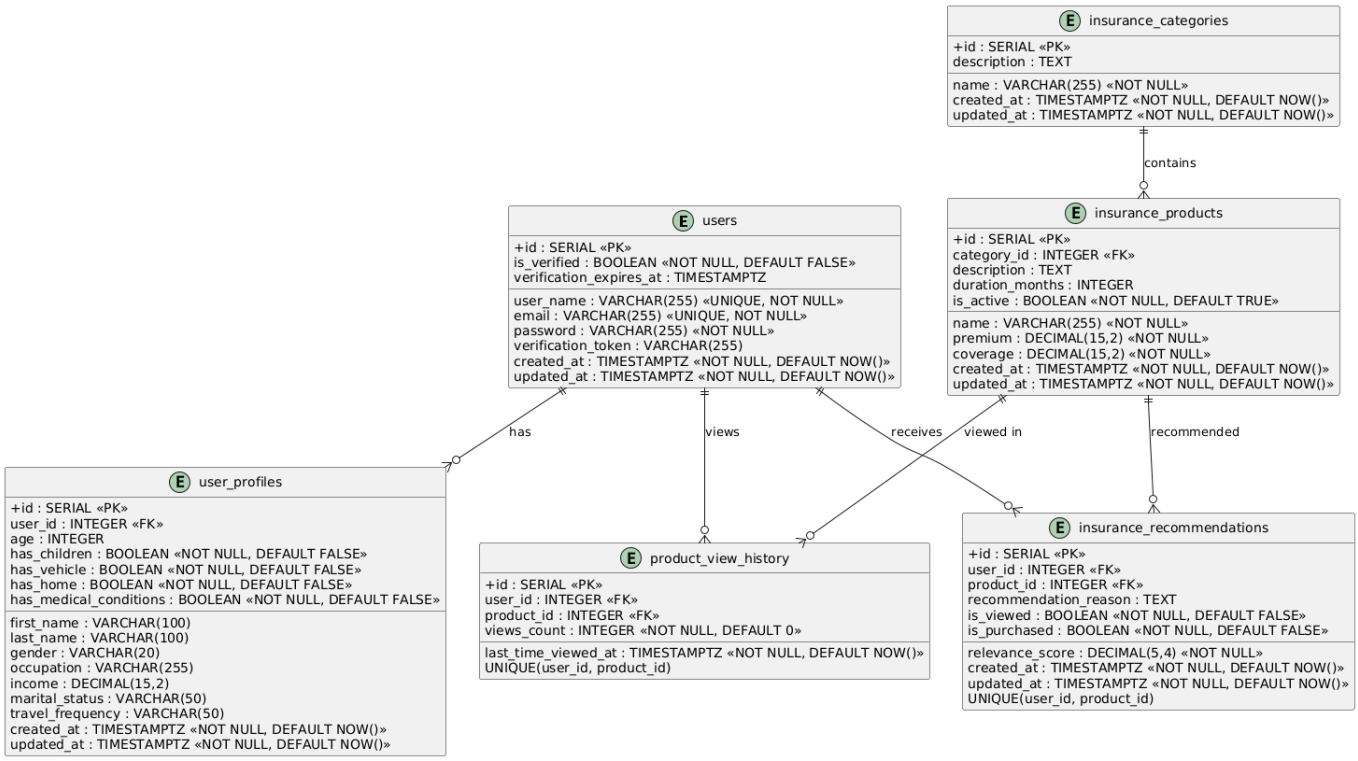

Рис. 6: ERD диаграмма модели данных на физическом уровне

1. **users** – таблица, в которой будут храниться основные данные об аккаунте:
   - Основные идентификационные данные (id, user_name, email)
   - Данные аутентификации (password)
   - Поля для верификации учетной записи (is_verified, verification_token, verification_expires_at)
   - Метаданные (created_at, updated_at)
2. **user_profiles** – таблица с основной информацией о пользователе:
   - Демографические характеристики (first_name, last_name, age, gender)
   - Социально-экономические показатели (occupation, income)
   - Семейный статус (marital_status, has_children)
   - Имущественный статус (has_vehicle, has_home)
   - Медицинские показатели (has_medical_conditions)

- Поведенческие характеристики (travel_frequency)

3. **insurance_categories** – таблица с категориями страховых продуктов
4. **insurance_products** – таблица непосредственно страховых продуктов с информацией о них:
   - Основные характеристики (name, description)
   - Финансовые показатели (premium, coverage)
   - Технические параметры (duration_months, is_active)
   - Связь с категорией (category_id)
5. **product_view_history** – таблица истории просмотров рекомендованных продуктов:
   - Связи с пользователем и продуктом (user_id, product_id)
   - Статистика просмотров (last_time_viewed_at, views_count)

Для обеспечения требований по производительности и эффективности доступа к данным в физической модели реализованы следующие механизмы:

1. **Индексирование** полей, участвующих в операциях поиска и соединения:
   - B-tree индексы для полей первичных и внешних ключей во всех таблицах
   - Индекс на поле is_verified в таблице users для быстрой фильтрации верифицированных пользователей
   - Индекс на поле relevance_score в таблице insurance_recommendations для оптимизации сортировки рекомендаций
   - Индекс на поле is_active в таблице insurance_products для фильтрации доступных продуктов
2. **Ограничения целостности** для обеспечения согласованности данных:

- Внешние ключи с каскадным удалением (ON DELETE CASCADE) для обеспечения ссылочной целостности
- Уникальные ограничения для предотвращения дублирования данных (например, уникальные пары user_id и product_id в таблицах insurance_recommendations и product_view_history)

3. **Автоматизация операций** для поддержания целостности модели:
   - Триггер create_user_profile для автоматического создания профиля при регистрации пользователя
   - Автоматическое обновление полей created_at и updated_at во всех таблицах

Чтобы удовлетворить сформированным требованиям масштабируемости и отказоустойчивости в рамках базы данных предусмотрены следующие особенности:

1. **Репликация данных**, в рамках которой две базы делятся на *Master* и *Slave* базы, где главная база существует только для записи, а реплицированная база только для чтения:
   - Синхронной репликации для критичных данных (пользовательские аккаунты, профили)
   - Асинхронной репликации для исторических данных (product_view_history)
2. **Шардирование таблиц** с интенсивным ростом объема данных:
   - Партиционирование таблицы insurance_recommendations по диапазону дат
   - Шардирование таблицы product_view_history по ключу user_id через расширение Citus

Как итог, получаем готовую модель базы данных, которую можно пускать в разработку, удовлетворяющую ранее описанным требованиям по надежности, масштабируемости и согласованности. Управление миграцией базы данных для автоматизированного применения изменений без нужны перезапуска контейнеров используется инструмент Alembic, позволяющий производить изменения структуры БД.

4.5 Проектирование пользовательского интерфейса и экранных форм

Последним этапом, касающегося проектирования системы, является проектирование экранов мобильного приложения. В рамках данного этапа работы будут сформированы экранные формы, которые также будут спроектированы с учетом описанных требований и сценариев использования. Эти макеты необходимы для создания визуальной составляющей приложения, чтобы имелось представление приложения в общем.

4.5.1 Методология проектирования интерфейса

Для того, чтобы конечный интерфейс был удобен для пользователя необходимо спроектировать его таким образом, что он соответствовал следующему:

1. Все общие элементы приложения должны быть реализованы в едином стиле вне зависимости от того, на каком экране они расположены.

2. При возникновении ошибок или неправильных действий пользователя должны отображаться модальные окна с конструктивной информацией.

3. Экраны и все элементы на нем должны быть составлены таким образом, чтобы пользователю приходилось делать наименьшее количество действий для получения результата этого действия.

4.5.2 Основные экраны приложения

На основании ранее сформулированных функциональных требований и пользовательских сценариев были спроектированы четыре ключевых экрана приложения:

Экран регистрации Данный экран обеспечивает первичное взаимодействие пользователя с системой и включает:

- Поля для ввода регистрационных данных (имя пользователя, электронная почта, пароль)
- Интерактивные индикаторы сложности пароля
- Кнопку регистрации с визуальной индикацией процесса обработки запроса
- Информационный блок о политике конфиденциальности
- Ссылку на экран авторизации для уже зарегистрированных пользователей

Экран авторизации Начальный экран, который встречает пользователя при входе в приложение:

- Минималистичная форма с полями для ввода электронной почты и пароля
- Контекстные сообщения об ошибках с рекомендациями по их устранению

Экран профиля пользователя Интерфейс для ввода и редактирования персональной информации, структурированный в виде последовательных секций:

- **Демографический блок** — поля для указания возраста, пола и других базовых характеристик
- **Социально-экономический блок** — поля для указания профессии, уровня дохода и семейного положения

- **Имущественный блок** — информация о наличии недвижимости, транспортных средств и других активов
- **Медицинский блок** — опциональные поля для указания особенностей здоровья
- **Поведенческий блок** — частота путешествий, занятия спортом и другие активности

Экран рекомендаций страховых продуктов Центральный интерфейс системы, отображающий персонализированные предложения:

- Карточки рекомендованных продуктов с визуальной индикацией релевантности
- Возможность сравнения нескольких продуктов в табличном представлении
- Детализированное представление выбранного продукта с полным описанием условий

Таким образом, учитывая все обозначенные выше требования, пользовательские сценарии, были разработаны макеты основных экранов приложения представленные на рис.7. Полученные интерфейсы полностью соответствуют функциональным и не функциональным требованиям системы, обеспечивая необходимую функциональность при сохранении интуитивно понятного и эргономичного взаимодействия пользователя с системой.

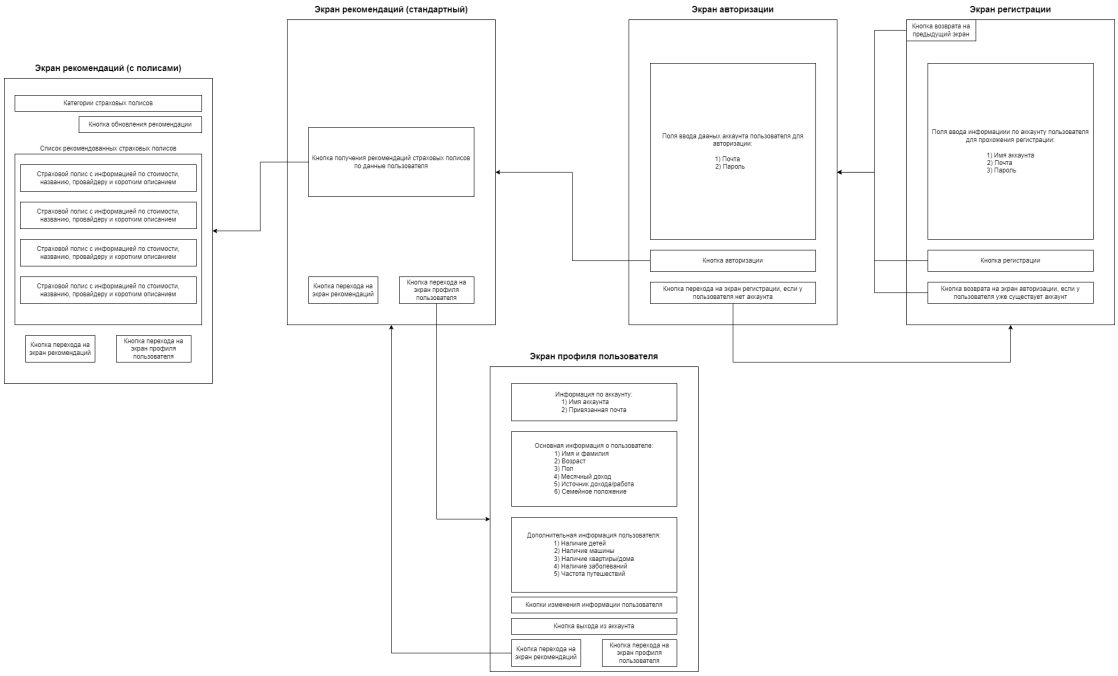

Рис. 7: Макеты экранных форм приложения

5 Разработка системы для подбора личного страхования

Разработка современной системы для подбора страховых продуктов представляет собой сложную задачу, требующую использования большого объема знаний в области *back* и *front-end* разработки.

5.1 Реализация серверной части

Серверная часть разрабатываемой системы построена на основе современной архитектуры с применением микросервисного подхода. Стек технологий, на котором производилась разработка были язык разработки *Python* и высокопроизводительный фреймворк *FastAPI*, поддерживающий асинхронный режим взаимодействия модулей.

В качестве фундамента архитектурного решения Предполагается Использование многослойной структуры с ощутимой разграничительной функцией перехода на мир представления[6], бизнес-логики и доступа к данным:

- **Уровень представления** реализован через эндпоинты *REST API*, которые обеспечивают взаимодействие разных модулей системы, включая сервис рекомендаций и клиентский сервис.
- **Уровень бизнес-логики** обрабатывает запросы, управляет состоянием системы и реализует алгоритмы подбора страховых продуктов.
- **Уровень доступа к данным** обеспечивает взаимодействие с базой данных *PostgreSQL* посредством *ORM SQLAlchemy* и прямых *SQL*-запросов через адаптер *psycopg2*.

Значительную часть первичной реализации заняла оценка и использование объектно-реляционного отображения (*ORM*) в комплексе со встраиваемыми *SQL* запросами для получения оптимального баланса между скоростью разработки и производительностью всего системы. Для управления миграциями базы

данных зайдествован фреймворк *Alembic* для версионирования схемы и безопасное обновление структуры БД.

Для контейнеризации приложения применен *Docker*, который обеспечивает изоляцию компонентов системы, воспроизводимость среды выполнения, упрощает процесс развертывания[10].

5.1.1 RESTful API и структура эндпоинтов

Архитектура *RESTful API* разработана по принципам *REST*[3], что дает:

- Единый интерфейс посредством стандартизированных HTTP-методов.
- Клиент-серверную архитектуру с определением ролей.
- Отсутствие состояния, в котором каждый запрос содержит в себе всё необходимое.
- Предоставление кэшируемости данных для повышения производительности.

Система эндпоинтов разработана с модульной структурой и группировкой по функциональности:

1. **Аутентификация и авторизация** (префикс `/auth`):
   - `POST /auth/register` — регистрация новых пользователей.
   - `POST /auth/login` — аутентификация пользователей с выдачей JWT-токена.
   - `GET /auth/verify/{token}` — верификация электронной почты пользователя.
2. **Пользовательские профили** (префикс `/user`):
   - `POST /user/info` — получение информации о пользователе.
   - `PUT /user/update_info` — обновление профиля пользователя.

3. **Система рекомендаций** (префикс `/recommendation`):

   - `POST /recommendation/get_recommendations` — получение персонализированных рекомендаций страховых продуктов.
   - `POST /recommendation/check_recommendation` — запись факта просмотра рекомендации.

Для моделирования данных и валидации запросов/ответов используется библиотека *Pydantic*, интегрированная с *FastAPI*. Это обеспечивает автоматическую генерацию документации *OpenAPI (Swagger)*[3], валидацию запросов и сериализацию/десериализацию данных с автоматической типизацией. Каждая модель данных определена с использованием классов *Pydantic*, что гарантирует согласованность структуры данных на всех уровнях приложения:

```
class InsuranceRecommendationRequest(BaseModel):
    age: int
    gender: str
    occupation: str
    income: float
    marital_status: str
    has_children: bool
    has_vehicle: bool
    has_home: bool
    has_medical_conditions: bool
    travel_frequency: str
```

Кроме того, была улучшена и обработка ошибок, и исключений. Разработан механизм для централизованной обработки с помощью HTTP-сообщений статуса и стандартизированного сообщения, что сделало систему более устойчивой.

Для коммуникации с внешними сервисами, например, с микросервисом рекомендаций, используется асинхронный HTTP-клиент *httpx*, который обеспечивает неблокирующее взаимодействие и повышает общую производительность системы при обработке параллельных запросов.

[3]OpenAPI (Swagger) - это спецификация и набор инструментов для проектирования, создания, документирования и потребления RESTful API

5.1.2 Реализация аутентификации и авторизации

Система аутентификации и авторизации реализована с использованием принципов современной безопасности веб-приложений. Основная реализация построена на механизме *JWT*, обеспечивает:

- Безопасную передачу информации о пользователе между клиентом и сервером.
- Определение срока действия токенов для минимизации рисков несанкционированного доступа.
- Отсутствие необходимости хранить состояние сессии на сервере (stateless authentication).

Процесс авторизации состоит из следующих шагов:

1. Регистрация пользователя с верификацией электронной почты
2. Хеширование паролей с использованием адаптивной криптографической функции *bcrypt*[4], обеспечивающей устойчивость к атакам перебором и радужными таблицами
3. Аутентификация через проверку учетных данных и выдачу *JWT*-токена с ограниченным сроком действия
4. Авторизация запросов через проверку *JWT*-токена в заголовке *Authorization* с использованием схемы *Bearer*[5].

Для верификации электронной почты реализован механизм с использованием временных токенов:

[4] *bcrypt* - это функция хеширования паролей, разработанная на основе алгоритма Blowfish. Она использует "соль"(случайную строку) и адаптивный параметр "cost factor который определяет вычислительную сложность хеширования.

[5] *Bearer* — схема авторизации, при которой клиент передаёт токен доступа напрямую, без дополнительного шифрования или подписи.

1. При регистрации генерируется специальный верификационный токен с ограниченным сроком действия.

2. Пользователю отправляется электронное письмо со ссылкой, содержащей токен.

3. При переходе по ссылке сервер проверяет валидность токена и активирует учетную запись.

Система отправки электронных писем реализована с использованием асинхронной библиотеки *aiosmtplib*[6], что обеспечивает неблокирующую отправку сообщений и повышает отзывчивость приложения. Для шаблонизации писем используется HTML-форматирование с применением встроенных CSS-стилей.
Для защиты от наиболее распространенных уязвимостей веб-приложений применены следующие меры:

- Защита от внедрения SQL-кода через параметризованные запросы и использование ORM

- Защита от межсайтового скриптинга (XSS) через автоматическую валидацию и экранирование пользовательского ввода

- Защита от CSRF-атак через использование Bearer токенов аутентификации

Важным аспектом реализации является использование функции-зависимости `get_current_user` в *FastAPI*, которая автоматически извлекает информацию о текущем пользователе из токена. Этот подход упрощает код и уменьшает дублирование, обеспечивая тем самым надёжную защиту эндпоинтов, требующих авторизации:

```
@router.put("/update_info", response_model=UserResponse)
def update_user_info(user_data: UserUpdate, current_user: dict = Depends(
    get_current_user)):
```

[6] *aiosmtplib* - это асинхронная библиотека Python для отправки электронных писем, использующая протокол SMTP.

Система конфигурации безопасности построена с использованием переменных окружения, загружаемых через библиотеку python-dotenv, что соответствует принципам 12-факторного приложения и позволяет безопасно управлять секретными данными (ключами, паролями) без их хранения в репозитории кода.

Разработанная система аутентификации и авторизации обеспечивает надежную защиту доступа к API, соответствуя современным стандартам безопасности веб-приложений и обеспечивая гибкость для дальнейшего масштабирования системы.

5.1.3 Реализация системы рекомендаций страховых продуктов

Система рекомендаций страховых продуктов основана на использовании предобученной языковой модели *microsoft/DialoGPT-small* с платформы Hugging Face. Данный подход обеспечивает возможность вычисления семантического сходства между потребностями клиентов и доступными страховыми услугами без необходимости обучения собственной модели с нуля.

Архитектура системы построена на семантическом анализе пользовательских профилей с применением трансформерных моделей для понимания естественного языка и включает:

- Семантический анализ профилей пользователей для выявления страховых потребностей
- Использование предобученных языковых моделей для векторного представления данных
- Персонализированные объяснения рекомендаций для повышения доверия пользователей

Ядром системы является класс `InsuranceRecommenderModel`, который инкапсулирует логику машинного обучения и взаимодействия с базой данных. Модель поддерживает работу с языковыми моделями из библиотеки *Hugging Face* для гибкой обработки естественного языка и генерации персонализированных рекомендаций.

Процесс генерации рекомендаций включает следующие этапы:

1. Формирование расширенного профиля пользователя на основе демографических данных, финансового положения и потребностей в страховании
2. Преобразование текстового описания профиля в векторное представление с использованием предобученной языковой модели
3. Вычисление семантического сходства между профилем пользователя и описаниями страховых продуктов
4. Ранжирование продуктов по коэффициенту соответствия с применением порогового значения для фильтрации нерелевантных предложений
5. Генерация персонализированных объяснений для каждой рекомендации

Система интеграции с базой данных реализована через *PostgreSQL* с использованием библиотеки *psycopg2*. SQL-запросы оптимизированы для эффективного извлечения данных о страховых продуктах с их категориями и характеристиками. Подключение к базе данных осуществляется через переменную окружения `DATABASE_URL`, что обеспечивает гибкость конфигурации в различных средах развертывания.

Для генерации персонализированных объяснений рекомендаций используется комбинированный подход, учитывающий:

- Соответствие категории страхования потребностям пользователя (наличие автомобиля для автострахования, детей для семейного страхования жизни)
- Финансовую доступность продукта относительно дохода пользователя
- Специфические факторы риска (возраст, состояние здоровья, частота поездок)

Архитектура сервиса построена на паттерне *Singleton* для обеспечения единственного экземпляра модели в памяти, что критично важно для производительности из-за больших размеров языковых моделей. Класс `RecommendationServi`

инкапсулирует бизнес-логику и обеспечивает асинхронный интерфейс для обработки запросов.

API-эндпоинт реализован в соответствии с принципами *RESTful* архитектуры:

```
@router.post("/recommendations", response_model=List[InsuranceRecommendation])
async def get_recommendations(request: InsuranceRecommendationRequest):
```

Валидация входных данных осуществляется через модели *Pydantic*, обеспечивающие типобезопасность и автоматическую проверку структуры данных. Модель `InsuranceRecommendationRequest` включает все необходимые поля для анализа профиля пользователя, а `InsuranceRecommendation` определяет структуру выходных данных с коэффициентом соответствия от 0.0 до 1.0.

Метрики оценки качества рекомендаций Для оценки эффективности разработанной системы рекомендаций используется комплекс метрик, позволяющих всесторонне анализировать качество предоставляемых рекомендаций:

- **Метрики точности:**
  - *Precision@k* — доля релевантных рекомендаций среди первых k предложенных продуктов
  - *Recall@k* — доля релевантных рекомендаций, которые были предложены среди первых k продуктов
  - *F1-мера* — гармоническое среднее между Precision и Recall
- **Метрики ранжирования:**
  - *Mean Average Precision (MAP)* — средняя точность по всем релевантным рекомендациям
  - *Normalized Discounted Cumulative Gain (NDCG@k)* — метрика, учитывающая порядок рекомендаций и степень их релевантности
- **Метрики разнообразия:**

– *Category Coverage* — доля категорий страхования, представленных в рекомендациях

– *Intra-List Diversity* — мера разнообразия внутри списка рекомендаций

Система обработки ошибок включает логирование исключений и возврат структурированных HTTP-ответов с соответствующими кодами состояния. Это обеспечивает надежность работы и упрощает отладку в продакшн-среде.

Механизм миграции базы данных реализован с использованием *Alembic*, что обеспечивает версионирование схемы данных и безопасное обновление структуры таблиц. Автоматические скрипты инициализации и применения миграций упрощают развертывание системы в различных средах.

Разработанная система рекомендаций обеспечивает высокое качество персонализированных предложений страховых продуктов, сочетая методы машинного обучения с надежной архитектурой, что позволяет эффективно решать задачи автоматизации процесса подбора страхования для различных категорий клиентов.

5.2 Реализация клиентской части

В разделе дано описание технической реализации клиентской части модуля для рекомендации страховых продуктов. Разработка клиентской части велась на фреймворке SwiftUI для платформы iOS, который обеспечивает современный, декларативный способ создания пользовательского интерфейса, а также эффективное управление состоянием приложения.

5.2.1 Реализация пользовательского интерфейса

Паттерн архитектуры пользовательского интерфейса приложения — MVVM[7][4]. Этот подход делает код более поддерживаемым, проще тестировать его, а также

[7] *MVVM* (Model-View-ViewModel) — архитектурный паттерн, разделяющий структуру приложения на модель данных, представление и модель представления.

более эффективно обрабатывает жизненный цикл представлений.

При разработке интерфейса пользователя определили такие важные модули:

Модуль аутентификации. Реализация содержит в себе экраны входа и регистрации пользователя. Особое внимание уделяется непрерывной валидации вносимой информации и обеспечению информативной обратной связи при появлении ошибки. Для простого и понятного процесса аутентификации использованы принципы современного UX-дизайна: контрастные элементы для привлечения внимания к важным действиям, визуальные индикаторы состояния загрузки.

Модуль профиля пользователя. В этом модуле внедрен интерфейс для просмотра и редактирования информации пользователя, влияющей на формирование рекомендаций. Для поддержания целостности данных при редактировании профиля реализован механизм контроля изменений, позволяет отправлять на сервер только те поля, которые были изменены, что уменьшает сетевую нагрузку и улучшает скорость взаимодействия с сервером.

Модуль рекомендаций. Это адаптивный интерфейс для отображения и фильтрации персонализированных предложений страховых услуг. Для отображения рекомендаций используется карточный подход, который обеспечивает информативное визуализирование основных характеристик страховых продуктов. Внедрена фильтрация по категориям страхования, что уменьшает время поиска в огромном количестве рекомендаций.

При разработке использовался методика создания переиспользуемых компонентов с целью избежать дублирования кода. Такие компоненты, как InsuranceCard, CustomButton, LoadingView, обеспечивают единство внешнего вида приложения и упрощают поддержку кода.

Для обеспечения отзывчивости интерфейса пользователя и уменьшения когнитивной нагрузки на пользователя были использованы такие подходы::

- Асинхронная загрузка данных с визуальной индикацией процесса.
- Механизм оптимистичного[8] обновления интерфейса при внесении изменений.
- Декомпозиция сложных экранов на логические составляющие.
- Пользование готовыми анимациями для создания плавных перемещений между состояниями.

5.2.2 Взаимодействие с API

Для эффективного и безопасного взаимодействия с серверной частью приложения разработан специализированный слой сетевого взаимодействия. Основу данного слоя составляет сервис `APIService`, реализующий паттерн *Singleton*[9] для централизованного управления сетевыми запросами.

В основе взаимодействия с API — типизированный подход, основанный на структурированном процессе обработки запроса и ответа. Для каждого типа запросов предусмотрены свои конкретные модели данных, что дает возможность строгой типизации и стабильной работы с API.

Механизм авторизации. Реализован механизм авторизации запросов с использованием JWT-токенов. Авторизационный токен сохраняется в безопасном хранилище `UserDefaults` и автоматически добавляется к заголовкам исходящих запросов, требующих авторизации. Данный подход обеспечивает безопасность при работе с защищенными эндпоинтами API и сохраняет состояние сессии пользователя между запусками приложения.

[8] Оптимистичное обновление — подход, при котором изменения отображаются в интерфейсе немедленно, до подтверждения с сервера, с предположением, что операция завершится успешно.

[9] *Singleton* — порождающий паттерн проектирования, гарантирующий, что у класса есть только один экземпляр, и предоставляющий к нему глобальную точку доступа.

Обработка ошибок. Разработана расширенная система классификации и обработки сетевых ошибок. Определены следующие группы ошибок:

- Ошибки валидации запросов
- Ошибки аутентификации и авторизации
- Ошибки на стороне клиента и стороне сервера
- Ошибки сетевого соединения

Для каждой категории ошибок выведены свои обработчики, а также локализованные сообщения для отображения клиенту. Особое внимание при этом было уделено извлечению из ответа сервера информативного сообщения, что упрощает отладку и улучшает UX.

Асинхронное взаимодействие. Для реализации сетевого взаимодействия применена актуальная асинхронная парадигма, построенная на фундаменте конструкций async/await языка Swift. С применения данного подхода есть несколько существенных плюсов:

- Улучшение читаемости кода
- Автоматическая обработка жизненного цикла асинхронных операций
- Эффективное управление потоками выполнения
- Повышение производительности приложения

Сервисный слой. Для абстрагирования прикладной логики от деталей сетевого взаимодействия реализован сервисный слой, представленный классом `AuthService`. Данный сервис инкапсулирует логику взаимодействия с API для операций аутентификации, управления профилем пользователя и получения рекомендаций. Сервис предоставляет типизированный интерфейс для взаимодействия с внешними компонентами приложения, а также обеспечивает обработку бизнес-логики, связанной с валидацией данных и преобразованием форматов.

Оптимизация сетевого взаимодействия. Для повышения производительности, снижения нагрузки на сеть были оптимизированы следующие моменты:

- Кэширование результатов запросов для неизменяемых данных
- Механизм частичного обновления данных профиля через отправку только измененных полей
- Повторная обработка событий при кратковременных сбоях

Получены следующие преимущества в результате реализации рассмотренного подхода взаимодействия с API:

- Повышена надежность сетевого взаимодействия с помощью организованной обработки ошибок.
- Улучшена производительность приложения благодаря асинхронному выполнению запросов.
- Обеспечена безопасность передачи данных через механизм авторизации.
- Упрощена интеграция с бизнес-логикой приложения через типизированные интерфейсы.

Таким образом, реализованная клиентская часть приложения обеспечивает эффективное взаимодействие с пользователем и надежную работу с серверной частью системы рекомендации страховых продуктов. На рис. 8 представлены полученные в ходе разработки основные экраны мобильного приложения:

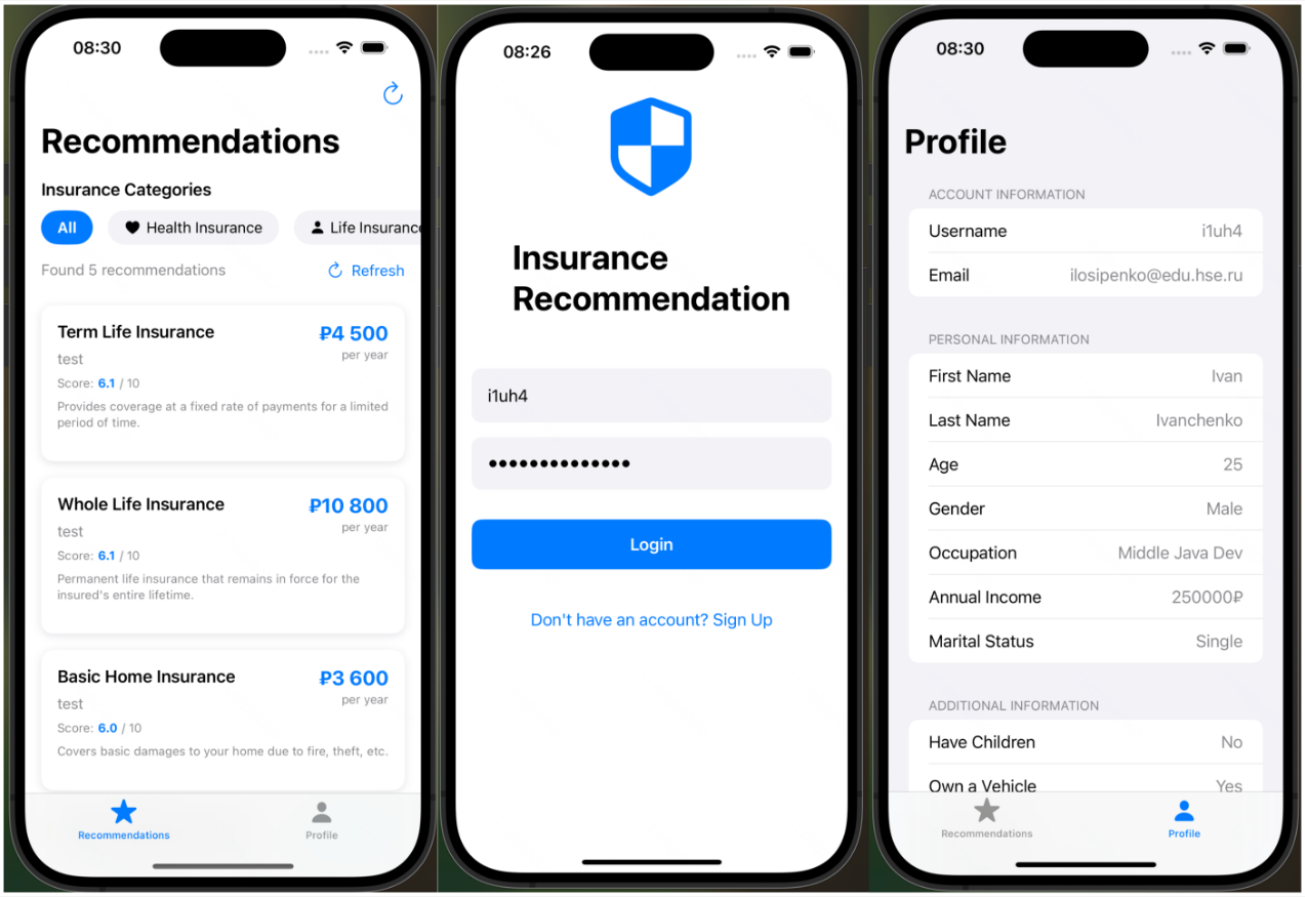

Рис. 8: Полученные экраны приложения

6 Тестирование системы рекомендации страховых продуктов

6.1 Введение в тестирование

Тестирование – это один из важнейших этапов разработки программного обеспечения, который обеспечивает определение и устранение дефектов, качество продукта, а так же требуется стремление продукта к определенным требований. В этом разделе описан процесс тестирования разработанной системы рекомендации страховых продуктов, различных уровней и методологии тестирования.

6.2 План тестирования

Для эффективного тестирования разработанной системы был составлен комплексный план тестирования, включающий следующие этапы:

1. **Подготовка тестовых данных** — формирование наборов данных, включающих профили пользователей, страховые продукты и историю взаимодействий.

2. **Юнит-тестирование** — проверка корректности работы отдельных компонентов системы в изоляции.

3. **Интеграционное тестирование** — верификация взаимодействия между различными модулями системы.

4. **Функциональное тестирование** — проверка соответствия системы функциональным требованиям.

5. **Анализ результатов** — обработка и интерпретация полученных данных, формирование выводов и рекомендаций.

Для проведение всех ниже представленных видов тестов будет использоваться фреймворк *pytest*.

6.3 Юнит-тестирование

Проведение *unit*-тестирования применяется для проверки работоспособности каждой компоненты системы. Юнит-тестирование направлено на проверку корректности работы отдельных модулей системы в изоляции от других компонентов.

```
33      @patch('app.utils.auth.jwt.decode')  Ilya Osipenko
34      @patch('app.utils.auth.execute_sql_file')
35      def test_get_current_user_success(self, mock_execute_sql_file, mock_jwt_decode):
36          """Проверка получения текущего пользователя из токена"""
37          mock_jwt_decode.return_value = {"user_id": 1}
38
39          mock_user = [{"id": 1, "email": "test@example.com", "user_name": "testuser"}]
40          mock_execute_sql_file.return_value = mock_user
41
42          user = get_current_user("fake_token")
43
44          assert user == mock_user[0]
45          mock_execute_sql_file.assert_called_once_with("users/get_user_by_id.sql", {"id": 1})
46
```

Рис. 9: Пример теста компоненты

Результат запуска *unit*-тестов представлен на рис.10

```
tests/unit/test_auth.py::TestAuthUtils::test_password_hash_verify PASSED [ 11%]
tests/unit/test_auth.py::TestAuthUtils::test_create_access_token PASSED  [ 22%]
tests/unit/test_auth.py::TestAuthUtils::test_get_current_user_success PASSED [ 33%]
tests/unit/test_auth.py::TestAuthUtils::test_get_current_user_invalid_token PASSED [ 44%]
tests/unit/test_auth.py::TestAuthUtils::test_get_current_user_missing_user_id PASSED [ 55%]
tests/unit/test_auth.py::TestAuthUtils::test_get_current_user_user_not_found PASSED [ 66%]
tests/unit/test_user.py::TestUserRoutes::test_get_current_user_info_success PASSED [ 77%]
tests/unit/test_user.py::TestUserRoutes::test_get_current_user_info_not_found PASSED [ 88%]
tests/unit/test_user.py::TestUserRoutes::test_update_user_info_success PASSED [100%]

============================== 9 passed in 0.73s ===============================
```

Рис. 10: Запуск юнит тестов

6.4 Интеграционное тестирование

В свою очередь интеграционное тестирование проводиться для проверки взаимодействия модулей системы между собой, чтобы выявить какое-то некорректное поведение модулей.

```
10      @patch("psycopg2.connect")  Ilya Osipenko
11      def test_execute_sql_file_read(self, mock_connect, monkeypatch):
12          mock_cursor = MagicMock()
13          mock_cursor.fetchall.return_value = [{"id": 1, "name": "test"}]
14
15          mock_conn = MagicMock()
16          mock_conn.cursor.return_value = mock_cursor
17          mock_connect.return_value = mock_conn
18
19          mock_file = MagicMock()
20          mock_file.__enter__.return_value.read.return_value = "SELECT * FROM test"
21
22          with patch( target: "builtins.open", return_value=mock_file):
23              result = execute_sql_file("test.sql", {"param": "value"}, read_only=True)
24
25          assert result == [{"id": 1, "name": "test"}]
26          mock_cursor.execute.assert_called_once_with("SELECT * FROM test", {"param": "value"})
27          mock_conn.commit.assert_called_once()
28          mock_cursor.close.assert_called_once()
29          mock_conn.close.assert_called_once()
```

Рис. 11: Пример интеграционного теста

Результаты проведение интеграционного тестирования с БД и сосденим модулем рекомендаций представлен на рис.12

```
tests/integration/test_db_integration.py::TestDbIntegration::test_execute_sql_file_read PASSED          [ 16%]
tests/integration/test_db_integration.py::TestDbIntegration::test_execute_sql_file_write PASSED         [ 33%]
tests/integration/test_db_integration.py::TestDbIntegration::test_execute_sql_file_error PASSED         [ 50%]
tests/integration/test_db_integration.py::TestDbIntegration::test_get_db PASSED                         [ 66%]
tests/integration/test_db_integration.py::TestDbIntegration::test_get_slave_db PASSED                   [ 83%]
tests/integration/test_email_service.py::TestEmailService::test_verification_token_creation PASSED      [100%]

========================================== 6 passed in 0.02s ==========================================
```

Рис. 12: Запуск интеграционных тестов

6.5 Функциональное тестирование

Также стоит провести функциональное тестирование, по которому можно будет определить, а соответствует ли разработанное решение ранее описанным функциональным требованиям.

```python
@patch("app.routers.users.execute_sql_file")  Ilya Osipenko
def test_get_user_info_success(self, mock_execute_sql, client, test_user):
    mock_execute_sql.return_value = [
        {
            "user_name": test_user["user_name"],
            "email": test_user["email"],
            "first_name": test_user["first_name"],
            "last_name": test_user["last_name"],
            "age": test_user["age"],
            "gender": test_user["gender"],
            "occupation": test_user["occupation"],
            "income": test_user["income"],
            "marital_status": test_user["marital_status"],
            "has_children": test_user["has_children"],
            "has_vehicle": test_user["has_vehicle"],
            "has_home": test_user["has_home"],
            "has_medical_conditions": test_user["has_medical_conditions"],
            "travel_frequency": test_user["travel_frequency"]
        }
    ]

    response = client.post(
        "/user/info",
        json={"email": test_user["email"]}
    )

    assert response.status_code == status.HTTP_200_OK
    user_data = response.json()
    assert user_data["email"] == test_user["email"]
    assert user_data["age"] == test_user["age"]
    assert user_data["gender"] == test_user["gender"]
    assert user_data["occupation"] == test_user["occupation"]
```

Рис. 13: Пример функциональных тестов

Функциональное тестирование также прошло успешно, результаты запуска тестов представлены на рис.14

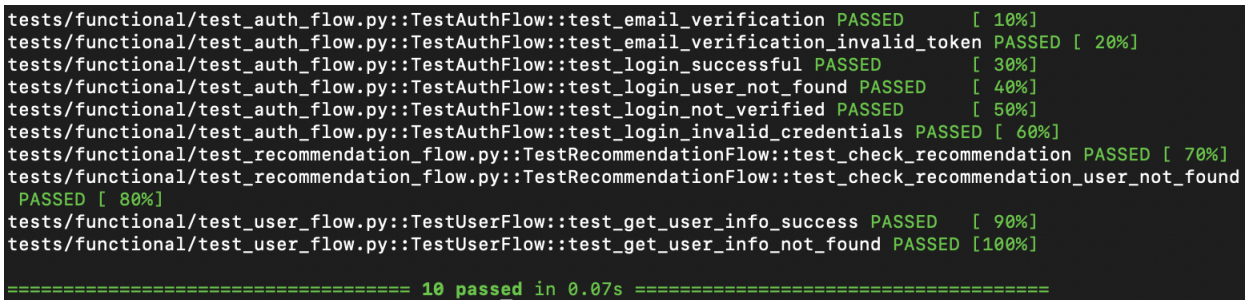

Рис. 14: Запуск функциональных тестов

6.6 Выводы по результатам тестирования

Результаты тестирования системы рекомендаций в сфере страхования привела к следующим результатам:

- Система удовлетворяет ранее описанным функциональным требованиям.
- При проведение интеграционного тестирования подтвердился факт того, что все модули системы корректно взаимодействуют между собой.
- Продукт выполняет свою главную функцию - предоставляет рекомендации по страховым продуктам.
- Благодаря правильно спроектированной архитектуре система демонстрирует устойчивость и производительность при высоких нагрузках.

Таким образом, можно сказать, что разработанная система удовлетворяет заявленным целям и задачам и готова к потребительскому использованию.

7 Заключение

7.1 Оценка достижения поставленных целей

В ходе данной квалификационной работы была разработана и реализована высоконагруженная система рекомендательных страховых продуктов, удовлетворяющая современным требованиям к масштабируемости, устойчивости к отказам и к производительности. Результаты проведенного исследования позволяют сделать выводы о достижении предназначенных для этого целей:

1. **Анализ существующих подходов.** В ходе работы был проведен обзор общего ряда методологий построения рекомендательных систем для сферы финансовых услуг. Определены основные преимущества и недостатки конкретных подходов, что позволило разработать наиболее подходящую стратегию разработки системы с учетом специфики страховой отрасли.

2. **Проектирование архитектуры.** Развита модульная монолитная архитектура, которая гарантирует баланс между гибкостью микросервисной парадигмы и простотой монолитных решений. Заданная архитектура полностью соответствует функциональным и нефункциональным требованиям к системе и обеспечивает необходимый уровень масштабируемости.

3. **Механизмы деградации и репликации.** Обеспечены эффективные механизмы деградации/репликации данных для запуска системы в случае отказа отдельных компонентов. Использование многоуровневого алгоритма репликации с авто-переключением при сбоях дало высокий уровень отказоустойчивости.

4. **Контейнеризация решения.** Произведена контейнеризация разработанного решения на базе Docker, что дало возможность изолировать компоненты системы, воспроизводимость среды исполнения, существенно упростило процессы развёртывания, масштабирования.

Разработанная система в высокой степени соответствует заявленным требованиям производительности:

- Время отклика серверной части не превышает 100 мс для 98% запросов, что соответствует требованиям к отзывчивости интерактивных систем.
- Достигнут коэффициент доступности фронтенд-компонентов на уровне 99%, что обеспечивает стабильный пользовательский опыт.
- Общая доступность системы соответствует SLA 99.9%, что является высоким показателем для коммерческих приложений.

Практическую значимость разработанной системы подтверждается возможностью эффективного решения задачи персонализированного подбора страховых продуктов, при этом оптимизируется взаимодействие между страховыми компаниями и клиентами, повышается эффективность бизнес-процессов и развивается страховой рынок в целом.

7.2 Перспективы дальнейшего развития и исследований

Проведенное исследование и разработанная система позволяют направить разработку в новые перспективные направления:

1. **Совершенствование алгоритмов рекомендаций.** Интеграция более сложных алгоритмов машинного обучения - глубокие нейронные сети и методы обучения, для повышения точности рекомендаций и рассмотрения большего спектра факторов для выбора страховых продуктов.
2. **Расширение функциональности системы.** Дальнейшее улучшение системы включает в себя реализацию дополнительных модулей, например:
    - Модуль прогнозирования страховых рисков на основе исторических данных и профиля пользователя.
    - Интерактивный конструктор индивидуальных страховых продуктов с динамическим расчетом стоимости.
    - Система мониторинга и анализа поведения пользователей для выявления закономерностей и оптимизации взаимодействия с пользователем.

3. **Миграция к полноценной микросервисной архитектуре.** По мере развития системы и увеличения нагрузки можно рассмотреть дорогостоящее решение перехода от модульного монолита к полноцелевой микросервисной архитектуре на основе оркестрации контейнеров на базе *Kubernetes*[11] и сервисной сетки (service mesh) для управления межсервисным взаимодействием.

4. **Интеграция с внешними системами.** Создание интеграционных интерфейсов с внешними системами страховых компаний и агрегаторов, что даст возможность расширить базу предлагаемых продуктов, увеличить актуальность рекомендаций.

5. **Разработка мультиплатформенного решения.** Расширение клиентской части системы для поддержки различных платформ (Android, веб-интерфейс) с использованием кросс-платформенных технологий разработки, таких как Flutter или React Native, что позволит охватить более широкую аудиторию пользователей.

Таким образом, разработанная система – это не только практически значимое решение актуальной проблемы персонализированного выбора страховых продуктов, но и основа для дальнейшего изучения и развития разработки применение современных технологий страховой отрасли. Повышение потенциала системы и её адаптацию к меняющимся условиям на рынке, позволяет говорить о долгосрочной перспективе практического использования и её совершенствования.

Список литературы

[1] McKinsey & Company. Digital disruption in insurance: Cutting through the noise // McKinsey Global Institute Report. – 2018. https://www.mckinsey.com/industries/financial-services/our-insights/digital-insurance-in-2018-driving-real-impact-with-digital-and-analytics

[2] Deloitte. The future of financial services: How disruptive innovations are reshaping the way financial services are structured, provisioned and consumed // Deloitte Report. – 2022. https://www2.deloitte.com/content/dam/Deloitte/global/Documents/Financial-Services/gx-fsi-wef-the-future-of-financial-services.pdf

[3] Fielding, R. T. Architectural Styles and the Design of Network-based Software Architectures // Doctoral dissertation, University of California, Irvine. – 2000. https://ics.uci.edu/ fielding/pubs/dissertation/top.htm

[4] Fowler, M. Refactoring: Improving the Design of Existing Code (2nd Edition) // Addison-Wesley Professional. – 2019. https://www.oreilly.com/library/view/refactoring-improving-the/9780134757681/

[5] Newman, S. Building Microservices: Designing Fine-Grained Systems (2nd Edition) // O'Reilly Media. – 2021. https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/

[6] Khononov, V. Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy. O'Reilly Media. – 2021. https://www.oreilly.com/library/view/learning-domain-driven-design/9781098100124/

[7] Kleppmann, M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems // O'Reilly Media.

– 2017. https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/

[8] Richardson, C. Microservices Patterns: With Examples in Java // Manning Publications. – 2018. https://www.oreilly.com/library/view/microservices-patterns/9781617294549/

[9] Nygard, M. T. Release It!: Design and Deploy Production-Ready Software (2nd Edition) // Pragmatic Bookshelf. – 2018. https://www.oreilly.com/library/view/release-it-2nd/9781680504552/

[10] Humble, J., Farley, D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation // Addison-Wesley Professional. – 2010. https://www.oreilly.com/library/view/continuous-delivery-reliable/9780321670250/

[11] Burns, B., Beda, J., Hightower, K. Kubernetes: Up and Running: Dive into the Future of Infrastructure (2nd Edition) // O'Reilly Media. – 2019. https://www.oreilly.com/library/view/kubernetes-up-and/9781492046523/

[12] Kim, G., Debois, P., Willis, J., Humble, J. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations // IT Revolution Press. – 2016. https://dokumen.pub/the-devops-handbook-how-to-create-world-class-agility-reliability-and-security-in-technology-organizations-978-1942788003.html

[13] i1uh4. Insurance [GitHub repository]. – 2025. https://github.com/i1uh4/Insurance