

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»
(Университет ИТМО)**

Факультет **Прикладной информатики**

Образовательная программа **Мобильные и сетевые технологии**

Направление подготовки **09.03.03 Прикладная информатика**

О Т Ч Е Т

**по производственной, технологической (проектно-технологической)
практике**

Тема задания: разработка LLM-ассистента для ответа на вопросы пользователей по эксплуатации сервиса: оптимизация промптов, запросов, векторизации и создание веб-интерфейса

Обучающийся: Хисаметдинова Динара Наилевна, группы К3341

Руководитель практики: Ходненко Иван Владимирович, к.т.н

Куратор практики: Улизько Максим Валерьевич

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	3
1 Анализ подходов к построению QA-систем	5
1.1 Расширение выборки — документации.....	5
1.2 Дообучение или адаптация эмбединговой модели	5
1.3 Традиционный метод поиска.....	6
1.4 Применение Retrieval Augmented Generation (RAG)	6
2 Развёртывание сервисов с разными LLM с целью поиска оптимальной	7
2.1 Saiga Llama 8B	7
2.2 Qwen/Qwen2.5-32B-Instruct-GPTQ-Int4.....	9
2.3 Сравнение двух моделей.....	9
3 Реализация и оптимизация поиска релевантных ответов	9
3.1 Постановка задачи и проблемы	9
3.2 Предобработка данных	10
3.3 Извлечение ключевых слов	10
3.4 Векторизация и индексирование	14
3.5 Уточнение пользовательского запроса и гибридный поиск	14
4 Проектирование взаимодействия будущих компонентов в контейнер-сервисе.....	16
5 Написание бэкенда на FastAPI для интеграции наиболее эффективной модели.....	18
6 Разработка фронтенд-части на Angular.....	20
7 Тестирование и развёртывание production варианта	20
7.1 Тестирование	20
7.2 Развёртывание	20
ЗАКЛЮЧЕНИЕ	23
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....	24

ВВЕДЕНИЕ

В Low-code решение на базе национального центра когнитивных разработок, лаборатории Интеллектуальных технологий принятия решений ежегодно внедряется новый функционал, делая платформу универсальным инструментом для автоматизации построения моделей технологических и бизнес-процессов с возможностью перебрасывания их между проектами, а также совместной работы. Особое внимание уделяется формированию ML-пайплайнов, где каждый этап обработки данных, от предварительной очистки и трансформации до обучения моделей и их оценки, представляется в виде интуитивно понятных узлов вычислительного графа. Такой подход позволяет гибко настраивать и оптимизировать процессы, а также визуально контролировать их выполнение, оценивать точность, менять гиперпараметры, что существенно повышает эффективность разработки и поддержки сложных моделей. За усложнением функционала последовало и разрастание документации, что увеличивает порог входа для начала работы с платформой. Новые возможности требуют от пользователя глубокого погружения в технические детали, понимания принципов работы узлов и взаимосвязей в графе, а также знания специфики обработки больших объемов данных. Это зачастую приводит к значительным временным затратам на изучение материалов и адаптацию к системе.

Для решения этой проблемы был разработан и развёрнут сервис вопросно-ответного (QA) LLM-ассистента (Large Language Model), отвечающий на вопросы пользователей по документации.

Задачи проекта:

- анализ существующих подходов к построению QA-систем, ознакомление с документацией существующей платформы,
- развёртывание сервисов с разными LLM с целью поиска оптимальной,
- реализация и оптимизация поиска релевантных ответов на пользовательский вопрос с векторизацией и уточнением запросов,

- проектирование взаимодействия будущих компонентов в контейнер-сервисе,
- написание бэкенда на FastAPI для интеграции наиболее эффективной модели,
- разработка фронтенд-части на Angular,
- тестирование и развёртывание production варианта.

1 Анализ подходов к построению QA-систем

1.1 Расширение выборки — документации

Был предоставлен json, в котором контекстом является документация, разбитая на 16 фрагментов (чанков) по смыслу, а также вопрос, который может возникнуть у пользователя по этой теме, а также соответствующий ему ответ. Он представлен на сниппете ниже.

Листинг 1 – Разметка контекста

```
{
  "<ID>": {
    "Context": "Текстовый контекст",
    "questions_list": [
      "Вопрос 1",
      "Вопрос 2",
      "..."
    ],
    "answers_list": [
      "Ответ 1",
      "Ответ 2",
      "..."
    ]
  }
}
```

Это вручную размеченный файл, в нём 800 строк, однако этого недостаточно для того, чтобы семантическое сходства вопроса пользователя и базы было высоким. Разбиение контекста на чанки важно, так как они позволяют укладываться в ограниченное контекстное окно модели, а также повышают эффективность индексации и работы с векторными хранилищами. Можно использовать LLM для генерации синонимичных вариантов исходного запроса, чтобы охватить различные формы написания (например, "Гугл" и "Google"). Это позволит расширить набор векторов, что повысит вероятность нахождения соответствия с имеющимися данными.

1.2 Дообучение или адаптация эмбединговой модели.

Можно дообучить модель на документации (инструкционный fine-

tuning). Далее продолжить обучение на данных, специфичных для IT, машинного обучения, бизнес-терминов, то есть осуществить fine-tuning под домен. Есть также необходимость исключить забывание старых знаний и правильно адаптировать огромный объём данных LLM к небольшому корпусу имеющихся в базе платформы. Однако, это требует слишком много вычислительных мощностей, которых нет на данном этапе.

1.3 Традиционный метод поиска.

Традиционный поиск в QA-системах базируется на статических алгоритмах ранжирования, таких как BM25 и TF-IDF, а также на индексировании с помощью векторных методов (например, библиотеки FAISS — Facebook AI Similarity Search, нужна для поиска ближайших соседей). Он включает токенизацию документа, расчет частот терминов и оценку релевантности запроса по заранее заданной формуле, что обеспечивает быстрый поиск, но может быть менее адаптивным к терминологическим расхождениям.

1.4 Применение Retrieval-Augmented Generation (RAG).

При поступлении запроса классический RAG-подход сначала применяет методы поиска для быстрого извлечения из документации наиболее релевантных фрагментов (чанков). Отбираются те, которые содержат информацию, соответствующую запросу.

Далее извлечённый контекст вместе с уточнённым запросом передается в LLM, которая, опираясь как на свои знания, так и на поданный контекст, формирует итоговый ответ. Такой подход позволяет компенсировать недостатки чисто поисковых систем, адаптируя генерацию к специфике документации. Для этого не нужно много вычислительных ресурсов, как в случае с дообучением, но из недостатков — увеличение времени на генерацию и сильная зависимость от качества поиска.

2 Развёртывание сервисов с разными LLM с целью поиска оптимальной

2.1 Saiga Llama3 8b

Для тестирования модели до предоставления доступа к серверу работодателя был развернут сервис с IlyaGusev/saiga_llama3_8b на CPU, однако при батчевой обработке запросов был использован transformers, код внутри модели выполнялся синхронно, даже при использовании async def, так как многие LLM фреймворки, в том числе transformers, не поддерживает асинхронные вызовы. Также эти библиотеки используют внутреннюю многопоточность, что может конфликтовать с ThreadPoolExecutor, а конкуренция за память приводит к Out of Memory. Таким образом, от этой идеи пришлось отказаться.

Далее был развёрнут сервис в Dev-container, однако, в нём нельзя установить Docker in docker, поэтому установка зависимостей производилась в venv. Файл api_gateway.py (листинг 2) поднимает FastAPI сервер, принимает запросы на эндпоинт (конечная точка веб-сервиса), генерирует task_id и отправляет задачу в очередь Redis и возвращает его в ответе.

Листинг 2 — api_gateway.py, запускающий RedisBroker

```
app = FastAPI()

class ChatRequest(BaseModel):
    user_input: str

async def process_request(prompt: str) -> str:
    payload = {
        "model": "IlyaGusev/saiga_llama3_8b",
        "prompt": prompt,
        "max_tokens": 512
    }
    try:
        async with httpx.AsyncClient(timeout=60) as client:
            response = await client.post(VLLM_API_URL,
```

```

        json=payload)
        response.raise_for_status()
        result = response.json()
        return result["choices"][0]["text"].strip()
    except httpx.RequestError as e:
        return f"Ошибка обработки запроса: {e}"
@app.post("/chat/")
async def chat(request: ChatRequest):
    task_id = str(uuid.uuid4())
    response_text = await process_request(request.user_input)
    return {"task_id": task_id, "response": response_text}

```

Представленный на листинге 3 воркер, или обработчик, (`worker.py`) подключается к Redis и ‘слушает’ очередь `task_queue`, получает сообщение из очереди, затем отправляет `prompt` в `vLLM`, получает сгенерированный большой языковой моделью результат. Работает асинхронно и запускается через `asyncio.run(FastStream(broker).run())`.

Листинг 3 — `worker.py`, запускающий `RedisBroker`

```

broker = RedisBroker("redis://10.32.15.90:6379")
class TaskRequest(BaseModel):
    task_id: str
    prompt: str
@broker.subscriber("task_queue")
async def process_task(task: str):
    task_id, prompt = task.split(":", 1)
    print(f"Получен: {task_id}")
    payload = {
        "model": "IlyaGusev/saiga_llama3_8b",
        "prompt": prompt,
        "max_tokens": 512
    }
    try:
        async with httpx.AsyncClient(timeout=60) as client:
            response = await client.post(VLLM_API_URL,
            json=payload)

```



```

        response.raise_for_status()
        result = response.json()
        output_text = result["choices"][0]["text"].strip()
        print(f"Результат для {task_id}: {output_text}")
    except httpx.RequestError as e:
        print(f"Ошибка обработки запроса {task_id}: {e}")
if __name__ == "__main__":
    asyncio.run(FastStream(broker).run())

```

2.2 Qwen/Qwen2.5-32B-Instruct-GPTQ-Int4

Для сравнения была развёрнута модель с 32 миллиардами параметров, но уже не дообученная на корпусе русскоязычных текстов. Процесс развёртывания идентичен тому, что был с моделью Saiga. Работа происходила на видеокартах NVIDIA RTX 6000 Ada, в Dev-container, предоставленном организацией.

2.3 Сравнение двух моделей

Был написан скрипт для тестирования эффективности выбранных LLM.

3 Реализация и оптимизация поиска релевантных ответов

3.1 Постановка задачи и проблемы

Было протестировано множество разных скриптов чтобы выявить тот, что выдаёт больший показатель косинусного сходства между вектором вопроса пользователя и вопросом из базы данных.

На рис. 1 представлена схема, показывающая, какую задачу нужно решить, где знаки вопроса

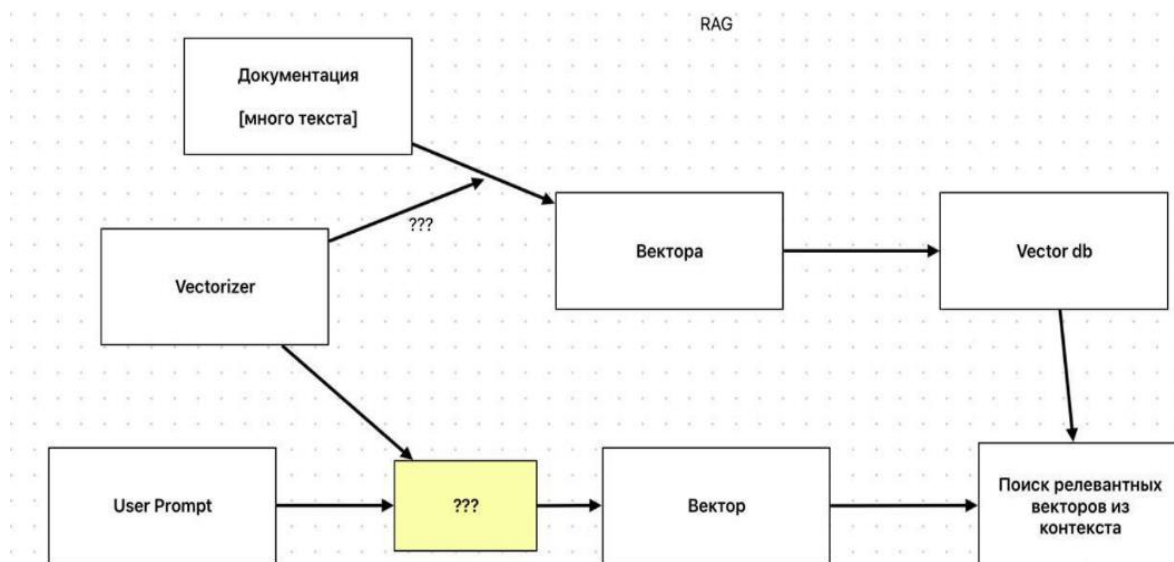


Рисунок 1 – Схема с постановкой задачи

Основные проблемы, выявленные при тестировании:

- несоответствие векторных представлений вопросов и ответов, что затрудняет их поиск в векторном пространстве,
- различие в терминологии между пользовательскими запросами и формулировками в документации,
- высокая вычислительная нагрузка при частом обращении к LLM для уточнения запросов.

Для решения этих проблем был применён гибридный подход к поиску, включающий BM25, FAISS и LLM.

3.2 Предобработка данных

Как было указано выше, данные были агрегированы в пары «вопрос—ответ» с сохранением контекста. Большой языковой моделью было сгенерировано ещё несколько похожих вопросов (с параметром температуры, то есть детерминированности, равным 0.1), с одинаковыми ответами из базы.

3.3 Извлечение ключевых слов

Для каждого вопроса из базы проводилось автоматическое выделение ключевых слов методами NLP (Natural Language Processing). После загрузки

данных они разбиваются на три массива: contexts (контекст документации), questions и answers. Они извлекались и векторизовались с использованием SentenceTransformer. Полученные эмбединги индексировались в FAISS, что позволило ускорить поиск на 32% по сравнению с последовательным сравнением векторов. Однако этот подход дал среднее косинусное сходство менее 0.4. Поскольку на этапе генерации статического JSON-файла (листинг 4) скорость обработки не является критически важной, была использована LLM, обеспечивающая большую точность терминов (Qwen 32B).

Листинг 4 – Пример ключевых слов, которые были сгенерированы

```
"3": {
    "question": "Какие поля обязательны для заполнения при
регистрации через стандартную форму?",
    "keywords": "обязательны, поля, регистрация, стандартная
форма"
},
"4": {
    "question": "Какие действия нужно выполнить после
регистрации через стандартную форму, чтобы подтвердить почту?",
    "keywords": "действия, регистрация, стандартная форма,
подтвердить почту"
},
"5": {
    "question": "Какие данные необходимо ввести для
авторизации после подтверждения почты?",
    "keywords": "данные, ввести, авторизации, подтверждение,
почты"
},
```

Вопросы обрабатывались пакетами по 10 штук, отправлялись в LLM с заданным промптом, а затем результаты очищались и сохранялись. В случае ошибок запросы повторялись до трех раз. Дополнительно был использован BM25+, который выполнял предварительный отбор кандидатов, сокращая объем выборки на 78% перед векторным поиском. Для корректной работы с асинхронными запросами использован nest_asyncio, так как работа производилась в irunb (листинг 5).

Температура модели для задачи выделения важных слов была выбрана 0.2, так как необходимо было сохранить баланс между точностью, чтобы не генерировались совсем лишние по смыслу слова, и некоторой степенью вариативности, так как при составлении базы вопросов человек мог подобрать не самую точную формулировку.

Листинг 5 – Инициализация данных и инструментов обработки запросов

```
with open("qna_dataset_smile_330_with_context.json", "r",
encoding="utf-8") as f:
    smile_qna = json.load(f)
os.environ["HUGGINGFACE_HUB_CACHE"] = "C:/huggingface_cache"
vectorizer = SentenceTransformer("sentence-transformers/all-mpnet-
base-v2")
vector_dim = vectorizer.get_sentence_embedding_dimension()
vector_db = faiss.IndexFlatL2(vector_dim)
questions = []
answers = []
contexts = []
for item in smile_qna.values():
    contexts.append(item["Context"])
    questions.extend(item["questions_list"])
    answers.extend(item["answers_list"])
nest_asyncio.apply()
```

Ниже приведена функция, генерирующая `processed_keywords.json` с ключевыми словами (листинг 6). При возникновении сетевой ошибки или тайм-аута выполняется повторная попытка запроса через 5 секунд. Если все попытки неудачны, в список `processed` добавляются исходные вопросы без обработки, чтобы избежать потерь данных.

После завершения обработки всех батчей соединение `requests.Session` закрывается, а функция возвращает список выделенных ключевых слов для всех переданных вопросов

Листинг 6 — Функция генерации ключевых слов

```
def batch_extract_keywords(questions, batch_size=10,
max_retries=3):
    processed = []
    session = requests.Session()
    for i in range(0, len(questions), batch_size):
        batch = questions[i:i+batch_size]
        batch_text = "\n".join([f"{idx+1}. {q}" for idx, q in
enumerate(batch)])
        data = {"user_input": f"Выдели ключевые слова для
следующих вопросов.\n" f"Выводи строго в формате: '1. ключевые
слова', '2. ключевые слова' и так далее.\n\n" f"{batch_text}"}
        headers = {"Content-Type": "application/json"}
        retry_count = 0
        while retry_count < max_retries:
            try:
                response = session.post(LLM_API_URL, json=data,
headers=headers, timeout=60)
                response.raise_for_status()
                batch_results = response.json().get("response",
"" ).split("\n")
                cleaned_results = [res.split(". ", 1)[1].strip()
if ". " in res else res.strip() for res in batch_results]
                processed.extend(cleaned_results)
                break
            except requests.exceptions.Timeout:
                retry_count += 1
                print(f"Таймаут (попытка
{retry_count}/{max_retries}). Повтор через 5 сек...")
                time.sleep(5)
            except requests.exceptions.RequestException as e:
                print(f"Ошибка при запросе к LLM: {e}")
                processed.extend(batch)
                break
    session.close()
    return processed
```

Далее, на рисунке 2 приведён тест, показывающий, что скрипт, принцип которого был описан выше, работает.

```

1 data = {
2     "model": "Qwen/Qwen2.5-32B-Instruct-GPTQ-Int4",
3     "messages": [
4         {
5             "role": "system",
6             "name": "system_assistant",
7             "content": "Ты NLP-модель, выделяющая ключевые слова."
8             "Выводи строго в формате: 'ключевое слово1, ключевое слово2, ключевое слово3' и так далее. "
9             "Не добавляй пояснений, только список ключевых слов."
10        },
11        {
12            "role": "user",
13            "name": "user_request",
14            "content": "Какие ключевые слова у фразы 'Что делать, чтобы модель работала?'"
15        }
16    ]
17 }
18
19 headers = {"Content-Type": "application/json"}
20 response = requests.post(LLM_API_URL, json=data, headers=headers, timeout=120)
21 response_data = response.json()
22 choices = response_data.get("choices", [])
23 if choices and "message" in choices[0] and "content" in choices[0]["message"]:
24     raw_answer = choices[0]["message"]["content"]
25     clean_keywords = raw_answer.strip()
26     print(f"response: {clean_keywords}")

```

{'response': 'модель, работать, делать'}

Рисунок 2 – Тест функции-генератора ключевых терминов

3.4 Векторизация и индексирование

Для быстрого поиска использовались различные модели векторизации:

- all-mpnet-base-v2 и multi-qa-mpnet-base-dot-v1 – показали высокое качество семантического поиска,
- all-MiniLM-L6-v2 – потребляла меньше ресурсов, но качество результатов было ниже

Алгоритм индексирования:

- BM25+ использовался для быстрого ранжирования совпадений,
- FAISS применялся для поиска ближайших соседей в векторном пространстве,
- Dense Passage Retrieval (DPR) и ColBERT тестировались, но были менее эффективны из-за увеличенной вычислительной нагрузки.

3.5 Уточнение пользовательского запроса и гибридный поиск

Были протестированы несколько скриптов. Для приведения запроса пользователя к терминологии документации использовалась LLM, что повысило точность поиска на 14%. Генерация синонимичных формулировок уменьшила среднее семантическое расстояние между запросом и данными на

18%, что повысило релевантность ответов. Однако этот этап увеличил время обработки на 2–15 секунд в зависимости от модели. Лучшим методом стало сочетание BM25+ и FAISS для поиска соседей в соотношении 70% на 30% соответственно.

Данный подход (листинг 7) позволил достичь среднего косинусного сходства между векторами пользовательских запросов и наиболее релевантными вопросами из базы более 0.87, а также ускорить процесс поиска на 32% по сравнению с полным перебором возможных кандидатов.

Листинг 7 — Оптимальный алгоритм поиска

```
def refine_query_with_llm(query):
    best_match = max(question_keywords.keys(), key=lambda q:
cosine_similarity(embed(query).reshape(1, -1), embed(q).reshape(1,
-1)) [0] [0])
    relevant_keywords = question_keywords.get(best_match, "")
    prompt = f"""\Ты помощник по документации платформы SMILE.
Переформулируй этот вопрос, добавив только релевантные
термины: {' '.join(relevant_keywords)}.\
Вопрос: "{query}"
Переформулированный запрос: ""
data = {"model": "Qwen/Qwen2.5-32B-Instruct-GPTQ-Int4",
        "messages": [{"role": "user", "content": prompt}]}
response = requests.post(LLM_API_URL, json=data).json()
refined_query = response.get("choices",
[{}])[0].get("message", {}).get("content", query)
return refined_query.strip().replace('""', '')

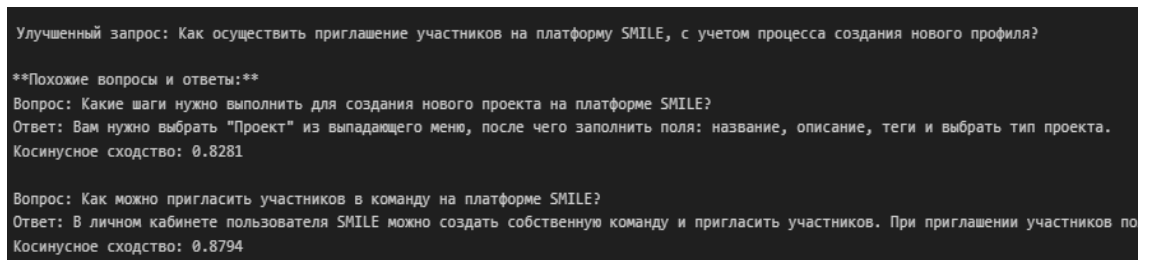
def find_similar_qna(query, top_n=2):
    refined_query = refine_query_with_llm(query)
    print(f"\Улучшенный запрос: {refined_query}")
    query_vector = embed(refined_query).reshape(1, -1)
    tokenized_query = refined_query.lower().split()
    bm25_scores = bm25.get_scores(tokenized_query)
    top_bm25_idx = np.argsort(bm25_scores)[-50:]
    _, faiss_indices = faiss_index.search(query_vector, 10)
    top_faiss_idx = faiss_indices[0]
    max_index = len(question_answer_pairs)
```

```

top_bm25_idx = [i for i in top_bm25_idx if i < max_index]
top_faiss_idx = [i for i in top_faiss_idx if i < max_index]
hybrid_scores = {
    i: 0.7 * bm25_scores[i] + 0.3 *
cosine_similarity(query_vector, question_vectors[i].reshape(1, -
1))[0][0]
    for i in range(len(question_vectors)) if i < max_index
}
best_indices = sorted(hybrid_scores, key=hybrid_scores.get,
reverse=True)[:top_n]
cos_similarities = [cosine_similarity(query_vector,
question_vectors[i].reshape(1, -1))[0][0] for i in best_indices]
filtered_results = [(question_answer_pairs[i][0],
question_answer_pairs[i][1], cos_similarities[idx]) for idx, i in
enumerate(best_indices) if cos_similarities[idx] > 0.65]
return filtered_results

```

После представленной части скрипта идёт функция `generate_answer_with_llm(context, query)`, генерирующая ответ через LLM используя контекст, если не находит достаточно схожих вопросов в базе данных. Результат выполнения кода приведён на рисунке 8.



```

Улучшенный запрос: Как осуществить приглашение участников на платформу SMILE, с учетом процесса создания нового профиля?

**Похожие вопросы и ответы:**
Вопрос: Какие шаги нужно выполнить для создания нового проекта на платформе SMILE?
Ответ: Вам нужно выбрать "Проект" из выпадающего меню, после чего заполнить поля: название, описание, теги и выбрать тип проекта.
Косинусное сходство: 0.8281

Вопрос: Как можно пригласить участников в команду на платформе SMILE?
Ответ: В личном кабинете пользователя SMILE можно создать собственную команду и пригласить участников. При приглашении участников по
Косинусное сходство: 0.8794

```

Рисунок 3 – Результат выполнения скрипта с лучшим косинусным сходством

4 Проектирование взаимодействия будущих компонентов в контейнер-сервисе

Все сервисы работают в одной docker-compose сети:

- frontend (Angular) отправляет POST /ask на Nginx, который проксирует их на backend (FastAPI),
- backend использует PostgreSQL для хранения вопросов и ответов,
- backend взаимодействует с LLM-сервисом, который ищет векторное представление запроса, уточняет его, ранжирует кандидатов из json

- для поиска самого близкого вопроса,
- база данных (PostgreSQL) хранит историю запросов,
- nginx (обратный прокси и раздача статических файлов фронтенда), также обеспечивает CORS-заголовки.

Контейнер chat-backend содержит Fastapi-приложение, обрабатывающее запросы, chat-frontend запускает Angular 19 с Vite. Nginx раздаёт фронтенд и проксирует API-запросы. Структура сервиса приведена на рисунке 4.

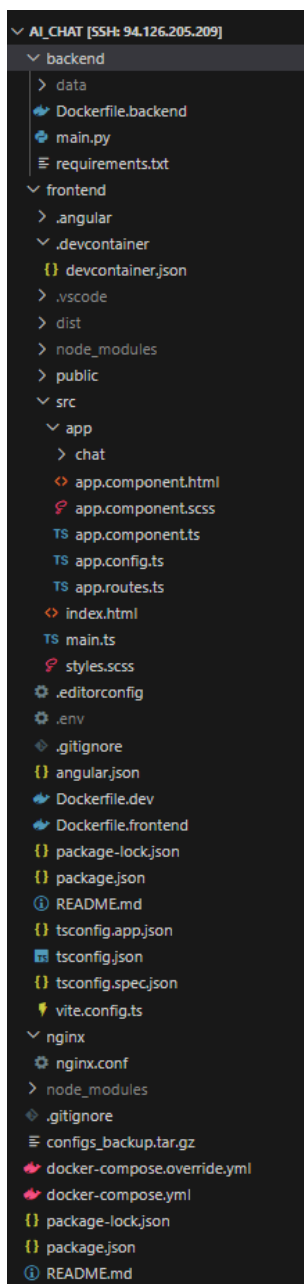


Рисунок 4 – Структура сервиса

5 Написание бэкенда на FastAPI для интеграции наиболее эффективной модели

Сначала загружаются конфигурационные параметры, включая информацию о базе данных, пути к файлам с предобработанными вопросами и ответы, а также URL API для взаимодействия с LLM. Далее создается объект FastAPI, к которому добавляется CORS-миддлварь, обеспечивающая поддержку кросс-доменных запросов. Настраивается подключение к базе данных с помощью SQLAlchemy, где создается engine, сессии и декларативная база.

При запуске приложения происходит инициализация базы данных: если необходимые таблицы отсутствуют, они создаются автоматически. Загружается обработанный корпус вопросов и ответов, а также список ключевых слов, соответствующих вопросам. Затем данные токенизируются и индексируются с использованием алгоритма BM25, что позволяет эффективно находить наиболее релевантные запросы на основе сходства с заданным вопросом.

При получении POST-запроса на эндпоинт /ask, сервер извлекает вопрос из тела запроса и проверяет, что он не пустой. Далее вызывается функция уточнения запроса с использованием LLM, которая переформулирует его для более точного поиска. Затем токенизированный вопрос используется в BM25 для поиска 10 наиболее схожих вопросов из базы. Эти кандидаты отправляются в LLM, который выбирает наиболее релевантный вариант.

После определения лучшего совпадения сервер ищет ответ на этот вопрос в предобработанной JSON-базе. Найденный ответ вместе с вопросом пользователя сохраняется в базе данных, чтобы обеспечивать дальнейший анализ и улучшение поиска. В конце сервер отправляет пользователю JSON-ответ, содержащий уточненный поисковый запрос, найденный наиболее похожий вопрос и соответствующий ему ответ.

Весь процесс основан на трех ключевых компонентах: асинхронном взаимодействии с базой данных, использовании BM25 для поиска и интеграции

с LLM для уточнения запроса и ранжирования кандидатов. Это позволяет системе эффективно обрабатывать запросы и предоставлять точные ответы пользователям. На листинге 8 приведена часть `main.py`, отвечающая за эндпоинт и точку входа.

Листинг 8 — API-эндпоинт для поиска ответа на пользовательский запрос

```
@app.post("/ask")
async def ask_question(data: dict, db: AsyncSession =
Depends(get_db)):
    user_query = data.get("question", "").strip()
    if not user_query:
        raise HTTPException(status_code=400, detail="Вопрос не
должен быть пустым")
    refined_query = await refine_query_with_llm(user_query)
    tokenized_query = refined_query.lower().split()
    bm25_scores = bm25.get_scores(tokenized_query)
    top_bm25_idx = np.argsort(bm25_scores)[-10:]
    best_candidates = [questions[i] for i in
reversed(top_bm25_idx)]
    best_match = await rerank_candidates_with_llm(user_query,
best_candidates)
    answer = find_answer(best_match)
    new_entry = QuestionAnswer(question=user_query, answer=answer)
    db.add(new_entry)
    await db.commit()
    return {"search_query": user_query, "matched_question":
best_match, "answer": answer}
@app.on_event("startup")
async def startup():
    await init_db()
```

6 Разработка фронтенд-части на Angular

Фронтенд взаимодействует с бэкендом через API, который проксируется через Nginx. Запрос отправляется методом POST с телом, содержащим введённый пользователем вопрос. API-адрес подставляется из переменной окружения, передаваемой через Vite, что позволяет адаптировать среду выполнения без изменения кода. После отправки запроса на бэкенд в интерфейсе активируется состояние загрузки, отображаемое пользователю.

Для быстрого тестирования клиентской части был создан `docker-compose.override.yml`, чтобы запускать на `localhost:4200`, а затем деплоить с помощью обычного `docker-compose.yml`.

7 Тестирование и развёртывание production варианта

7.1 Тестирование

Тестирование включало в себя проверку корректности работы фронтенда, бэкенда, взаимодействия с базой данных, а также интеграцию с LLM-сервисом и обработку запросов через Nginx.

Бэкенд тестировался с использованием HTTP-запросов через Postman и curl. Проверялась обработка корректных и некорректных запросов, взаимодействие с базой данных, а также корректность работы механизмов уточнения и ранжирования запросов с помощью LLM.

7.2 Развёртывание

Перед развёртыванием на сервере проверялась доступность необходимых портов, так как Nginx использует 80 для доступа к фронтенду, а бэкенд слушает запросы на 8010. VPN-соединение необходимо для доступа к LLM-API, и его корректная работа тестируется перед запуском системы. В `nginx.conf` были указаны настройки веб-сервера — путь до директории, где находятся статические файлы, установка версии HTTP, указание допустимых заголовков

клиента и их передача, разрешение запросов ото всех источников, обработка CORS-запросов, определение максимально допустимого количества соединений на один процесс.

После успешного поднятия контейнеров проводится финальная проверка работы сервиса. В браузере тестируется пользовательский интерфейс, отправляются тестовые вопросы, проверяется работа бэкенда и интеграция с базой данных. Дополнительно анализируются логи контейнеров на наличие ошибок, а также проверка правильности портов (рис. 10)

Name	Command	State	Ports
ai_chat_chat-backend_1	uvicorn main:app --host 0. ...	Up	
ai_chat_chat-frontend_1	/docker-entrypoint.sh nginx ...	Up	80/tcp
ai_chat_db_1	docker-entrypoint.sh postgres	Up	0.0.0.0:5432->5432/tcp,:::5432->5432/tcp
ai_chat_nginx_1	/docker-entrypoint.sh nginx ...	Up	0.0.0.0:80->80/tcp,:::80->80/tcp

Рисунок 5 – Проверка портов

Ниже приведён итоговый результат работы сервиса с корректным ответом на вопрос пользователя по документации.

Что вы хотите уточнить по работе сервиса?

как авторизоваться через Яндекс?

Узнать

Результат:

Ваш запрос: как авторизоваться через Яндекс?

Найденный вопрос: Какие действия необходимо выполнить для авторизации через социальную сеть Яндекс?

Ответ: Для авторизации через социальную сеть Яндекс необходимо нажать на кнопку Яндекс, которая находится на странице входа в SMILE. После этого вы попадете на платформу и сможете начать работу.

Рисунок 6 – Конечный результат с корректным ответом

Далее для примера были написаны различные вопросы, которые могли бы заинтересовать пользователя, протестирована релевантность, формат и

правильность записи данных в базу данных и продемонстрирована на рисунке 7 в виде мониторинга логов.

```
chat-backend_1 | 2025-03-04 03:39:21,996 INFO sqlalchemy.engine.Engine [cached since 2587s ago] ('Как добавить выбранные узлы Baseline?', 'Для добавления выбранных узлов Baseline на графе в модуль на платформе SMILE необходимо выбрать необходимые узлы в модальном окне, нажать кнопку "Применить" и перенести их на граф.')
chat-backend_1 | 2025-03-04 03:39:21,998 INFO sqlalchemy.engine.Engine COMMIT
chat-backend_1 | INFO: 192.168.208.5:45408 - "POST /ask HTTP/1.1" 200 OK
nginx_1 | 45.89.111.84 - - [04/Mar/2025:03:39:22 +0000] "POST /ask HTTP/1.0" 200 583 "http://94.126.205.209/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/133.0.0.0 Safari/537.36"
chat-backend_1 | 2025-03-04 03:39:42,550 INFO sqlalchemy.engine.Engine BEGIN (implicit)
chat-backend_1 | 2025-03-04 03:39:42,551 INFO sqlalchemy.engine.Engine INSERT INTO questions_answers (question, answer) VALUES ($1::VARCHAR, $2::VARCHAR) RETURNING questions_answers.id
chat-backend_1 | 2025-03-04 03:39:42,551 INFO sqlalchemy.engine.Engine [cached since 2608s ago] ('Как авторизоваться через Яндекс?', 'Для авторизации через социальную сеть Яндекс необходимо нажать на кнопку Яндекс, которая находится на странице входа в SMILE. После этого вы попадете на платформу и сможете начать работу.')
chat-backend_1 | 2025-03-04 03:39:42,553 INFO sqlalchemy.engine.Engine COMMIT
chat-backend_1 | INFO: 192.168.208.5:56244 - "POST /ask HTTP/1.1" 200 OK
nginx_1 | 45.89.111.84 - - [04/Mar/2025:03:39:42 +0000] "POST /ask HTTP/1.0" 200 604 "http://94.126.205.209/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/133.0.0.0 Safari/537.36"
chat-backend_1 | 2025-03-04 03:40:10,652 INFO sqlalchemy.engine.Engine BEGIN (implicit)
chat-backend_1 | 2025-03-04 03:40:10,653 INFO sqlalchemy.engine.Engine INSERT INTO questions_answers (question, answer) VALUES ($1::VARCHAR, $2::VARCHAR) RETURNING questions_answers.id
chat-backend_1 | 2025-03-04 03:40:10,653 INFO sqlalchemy.engine.Engine [cached since 2636s ago] ('Где найти детальное описание и документацию по выбранной модели?', 'Пользователь может найти более детальное описание и документацию по выбранной модели на платформе на странице модели.')
chat-backend_1 | 2025-03-04 03:40:10,654 INFO sqlalchemy.engine.Engine COMMIT
chat-backend_1 | INFO: 192.168.208.5:48782 - "POST /ask HTTP/1.1" 200 OK
```

Рисунок 7– Лог обработки данных на сервере, тест релевантности ответов

Ниже представлен QR-код для доступа на сайт, так как сервис работает только с вопросами, связанными с документацией платформы, лучше тестировать на предложенных в логах на рисунке выше вопросах.

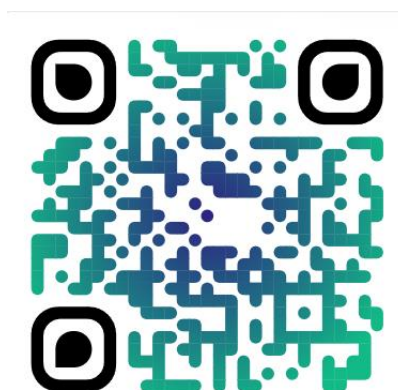


Рисунок 8– QR-код для доступа

ЗАКЛЮЧЕНИЕ

В результате выполнения производственной практики был успешно разработан и развернут сервис вопросно-ответного (QA) LLM-ассистента, призванный упростить взаимодействие пользователей с документацией Low-code платформы Национального центра когнитивных разработок.

В рамках проекта были решены все поставленные задачи:

- проведен анализ подходов к построению QA-систем,
- протестированы различные LLM для выбора наиболее эффективной модели в контексте ответов на вопросы по документации,
- разработан механизм поиска релевантных ответов,
- спроектирована архитектура взаимодействия всех компонентов в контейнеризированной среде,
- реализован бэкенд на FastAPI, обеспечивающий обработку запросов, взаимодействие с базой данных и интеграцию с LLM,
- создан фронтенд на Angular с удобным пользовательским интерфейсом для взаимодействия с системой,
- проведено тестирование сервиса и выполнено его развертывание в production.

Разработка данного ассистента показала эффективность использования генеративных моделей для навигации по сложной документации, что позволило снизить порог входа в платформу для новых пользователей.

В ходе работы была спроектирована архитектура, включающая взаимодействие frontend, backend, базы данных и модели LLM в Docker-контейнерах, были получены опыт в развёртывании сервисов и практика использования брокеров сообщений, библиотек для Data Science, а также таких фреймворков, как FastAPI, Angular.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Lewis P., Perez E., Piktus A., et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks: препринт [Электронный ресурс] // arXiv.org. — 2020. — URL: <https://arxiv.org/abs/2005.11401> (дата обращения: 06.02.2024).
2. Brown T., Mann B., Ryder N., et al. Language Models are Few-Shot Learners: препринт [Электронный ресурс] // arXiv.org. — 2020. — URL: <https://arxiv.org/abs/2005.14165> (дата обращения: 09.02.2024).
3. Devlin J., Chang M.-W., Lee K., Toutanova K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding: препринт [Электронный ресурс] // arXiv.org. — 2019. — URL: <https://arxiv.org/abs/1810.04805> (дата обращения: 09.02.2024)
4. Документация Redis // URL: <https://redis.io/docs/latest/> [Электронный ресурс] (дата обращения: 10.02.2024).
5. Raffel C., Shazeer N., Roberts A., et al. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer: препринт [Электронный ресурс] // arXiv.org. — 2020. — URL: <https://arxiv.org/abs/1910.10683> (дата обращения: 11.02.2024)