

Санкт-Петербургский государственный университет

Кафедра информатики

Группа 23.Б16-мм

# Реализация автоматизированной системы парсинга данных на основе больших языковых моделей

*Мурадян Денис Степанович*

Отчёт по учебной практике  
в форме «Решение»

Научный руководитель:  
старший преподаватель кафедры информатики, Бушмелев Федор Витальевич

Консультант:  
аналитик данных ПАО "Сбербанк России", Попов Артем Петрович

Санкт-Петербург  
2025

# Оглавление

<b>Введение</b>	<b>3</b>
<b>1. Постановка задачи</b>	<b>4</b>
<b>2. Обзор</b>	<b>5</b>
2.1. Методы парсинга веб-страниц . . . . .	5
2.2. Используемые технологии . . . . .	6
2.3. Выводы . . . . .	8
<b>3. Описание решения</b>	<b>9</b>
3.1. Получение и предварительная обработка веб-страницы . . . . .	9
3.2. Стратегии очистки HTML-разметки . . . . .	10
3.3. Интеграция с LLM: режимы <i>Structuring</i> и <i>Codegen</i> . . . . .	11
3.4. Механизм кэширования и семантического поиска запросов . . . . .	13
3.5. Пользовательские интерфейсы . . . . .	14
<b>4. Эксперимент</b>	<b>16</b>
4.1. Условия эксперимента . . . . .	16
4.2. Исследовательские вопросы . . . . .	17
4.3. Метрики . . . . .	18
4.4. Протокол измерений . . . . .	18
4.5. Результаты . . . . .	19
4.6. Примеры JSON-ответов . . . . .	19
4.7. Обсуждение результатов . . . . .	21
<b>5. Заключение</b>	<b>23</b>
<b>А. Приложение</b>	<b>25</b>
А.1. Примеры промптов и логика их формирования . . . . .	25
А.2. Пример промта для режима <i>Structuring</i> . . . . .	25
А.3. Пример промта для режима <i>Codegen</i> . . . . .	26
А.4. Логика подсказок ( <i>HintGen</i> ) . . . . .	28
А.5. Описание обрезки «лишнего» HTML . . . . .	29
<b>Список литературы</b>	<b>31</b>

# Введение

Веб-пространство сегодня характеризуется высокой технологической гетерогенностью: существуют как простые статические HTML-страницы [18], так и сложные одностраничные приложения (SPA), построенные с использованием React, Vue и других JavaScript-фреймворков [24]. Структура DOM, расположение элементов и способы загрузки данных могут существенно различаться от проекта к проекту [3]. В таких условиях создание универсального парсера, способного корректно извлекать данные с произвольного сайта, оказывается практически невыполнимой задачей. Разработчик вынужден тратить значительное время на анализ HTML-разметки, подбор селекторов и адаптацию к особенностям каждой конкретной страницы, а при изменении вёрстки — постоянно обновлять написанный код [19].

Это стимулирует поиск альтернативных способов извлечения данных. В последние годы большие языковые модели (LLM) продемонстрировали впечатляющие результаты в задачах анализа и генерации текстов [9, 7]. В контексте парсинга веб-страниц их применение позволяет реализовать автоматизированную систему, способную адаптироваться к различным структурам HTML-страниц и генерировать код для извлечения необходимых данных [8].

В данной работе используются два основных концептуальных подхода с LLM:

- *Structuring*: на вход модели подаётся очищенный HTML в сочетании с описанием требуемых полей, и в ответ модель возвращает готовую извлечённую информацию в формате JSON [9].
- *Codegen*: модель генерирует программный код на Python, который затем выполняется локально для извлечения данных с конкретного сайта [5].

Однако обращение к LLM для каждого нового запроса требует значительных вычислительных ресурсов и занимает длительное время [12]. В настоящей работе предложена архитектура с кэшированием: при использовании подхода Codegen каждое сгенерированное решение сохраняется в базу данных (SQLite и ChromaDB) [17]. При повторных запросах к тому же ресурсу повторное обращение к LLM не выполняется, а используется уже готовый скрипт, что позволяет существенно ускорить процесс извлечения данных.

# 1. Постановка задачи

Целью настоящей работы является ускорение сбора данных из открытых веб-источников за счёт автоматизации написания парсеров веб-страниц с помощью больших языковых моделей (LLM). Для достижения этой цели решаются следующие задачи:

1. Разработка модуля получения и предварительной обработки веб-страницы, способного определять необходимость JS-рендеринга и загружать как статические, так и динамически генерируемые (SPA) ресурсы (см. раздел 3.1).
2. Реализация стратегий очистки HTML-разметки: «полная» очистка, получающая чистый текст, и «лёгкая» очистка, сохраняющая базовую структуру документа (см. раздел 3.2).
3. Построение механизма интеграции с большими языковыми моделями, объединяющего генерацию кода парсера (Codegen) и кэширование результатов (SQLite и ChromaDB) для повторного использования без повторных обращений к LLM (см. раздел 3.3).
4. Проведение экспериментальной валидации разработанного решения: оценка корректности извлечения данных и измерение производительности в режимах «холодного» (cold) и «тёплого» (warm) запуска парсеров, а также сравнение с подходом Structuring (см. раздел 4).

## 2. Обзор

В данном разделе рассмотрены основные подходы и технологии, применяемые для извлечения данных с веб-страниц, а также существующие в литературе методы, близкие по назначению к задачам автоматизированного парсинга. Описаны их преимущества и недостатки с точки зрения универсальности, надёжности и затрат вычислительных ресурсов.

### 2.1. Методы парсинга веб-страниц

Существует два классических метода парсинга веб-страниц: *статический* и *динамический*.

**Статический парсинг** предполагает получение HTML-исходников напрямую через HTTP-запросы. Данный подход широко используется благодаря простоте реализации и высокой скорости работы. Типовой стек включает:

- **Requests** — библиотека для отправки HTTP-запросов[18].
- **BeautifulSoup4**[20] — парсер HTML/XML, позволяющий строить дерево тегов, искать элементы по селекторам и извлекать текст. Этот метод эффективен для страниц, контент которых формируется на сервере, но не подходит для SPA[3].

Преимущества статического подхода:

- высокая производительность (ограничивается временем сетевых запросов и обработкой текста)[19];
- простота отладки и воспроизводимость результатов при неизменном HTML.

Недостатки:

- неспособность корректно обрабатывать страницы с интенсивным JavaScript[2];
- необходимость ручного написания селекторов для каждой новой страницы.

**Динамический парсинг** предполагает эмуляцию браузера и выполнение JavaScript для получения «отрендеренного» DOM. К популярным инструментам относится:

- **Selenium**[2] — фреймворк для автоматизации браузера, позволяющий запускать Chrome/Firefox в фоновом режиме, ждать загрузки страницы и затем извлекать итоговый HTML. Такой метод подходит для современных SPA, но более медленный и ресурсоёмкий по сравнению со статическим парсингом[21].

Преимущества:

- возможность корректной обработки JavaScript-контента;
- высокая точность извлечения данных с динамических страниц.

Недостатки:

- значительные накладные расходы на запуск браузера и ожидание рендеринга;
- зависимость от версий браузерных движков и драйверов[14].

**Подходы на основе языковых моделей (LLM)** приобрели популярность после публикации работы Brown et al.[9], где продемонстрированы возможности LLM в режиме «few-shot». В контексте парсинга LLM применяются в двух основных режимах:

- *Structuring* (структурирование) — LLM получает на вход очищенный HTML и текстовое описание требуемых полей, после чего возвращает готовую JSON-структуру[7, 10].
- *Codegen* (генерация кода) — LLM формирует фрагмент Python-скрипта, который затем выполняется локально для извлечения данных. Такой подход сочетает гибкость LLM и возможность дальнейшей повторной работы без дополнительных запросов к модели[5].

Преимущества LLM-подходов:

- высокая адаптивность к разнообразным HTML-структурам;
- возможность «понимания» семантики страницы без явного знания её DOM-структуры[8].

Недостатки:

- необходимость вычислительных ресурсов и ограничение по числу токенов при обращении к облачным API[12];
- задержки при получении ответа от модели (latency);
- нестабильность результатов в «zero-shot» условиях.

## 2.2. Используемые технологии

Ниже приведён перечень ключевых технологий, используемых для реализации системы парсинга.

## Язык программирования

- **Python**[6] — выбран за счёт развитой экосистемы для веб-разработки, парсинга HTML и взаимодействия с LLM, а также встроенной поддержки SQLite.

## HTTP-запросы и статический парсинг

- **Requests**[18] — инструмент для отправки HTTP-запросов в Python.
- **BeautifulSoup4**[20] — библиотека для парсинга HTML и XML.

## Динамический парсинг

- **Selenium**[2] — фреймворк для автоматизации браузера, позволяющий получать «отрендеренный» HTML.

## LLM-интеграция

- **MistralAI (mistralai Python SDK)**[11] — клиентская библиотека для работы с LLM Mistral.
- **tiktoken**[13] — утилита для подсчёта токенов, оптимизирующая стоимость запросов к LLM.
- **Brown et al.**[9] — базовая статья, демонстрирующая потенциал LLM в режиме «few-shot».

## Хранение кэша

- **SQLite (sqlite3)**[1] — встраиваемая SQL-база для хранения метаданных и кэшированных скриптов.
- **ChromaDB**[4] — гибридная векторная база для семантического поиска.
- **Sentence-BERT**[17] — модель для преобразования текстовых запросов в векторное пространство.

## Пользовательские интерфейсы

- **FastAPI**[16] — фреймворк для разработки REST-API.
- **Gradio**[23] — библиотека для быстрой разработки веб-интерфейсов, развёртываемых в Hugging Face Space.
- **Jinja2**[15] — шаблонизатор для генерации HTML-страниц в веб-приложении.

## 2.3. Выводы

Из анализа существующих решений следует, что статический парсинг (Requests + BeautifulSoup4) эффективен для простых HTML-страниц, но не справляется с динамическими приложениями, требующими JavaScript (см.[2]). Применение LLM (см.[9]) открывает новые возможности автоматизации, однако требует значительных ресурсов и времени (см.[12]). Для создания универсального решения оправдан комбинированный подход, сочетающий традиционные методы парсинга с LLM-интеграцией и кэшированием, что позволяет оптимизировать вычислительные расходы и обеспечить повторный доступ к ранее сгенерированным парсер-скриптам.



### 3. Описание решения

Перед тем как подробно рассмотреть реализацию, приведём схематичный алгоритм работы системы:

1. Получение веб-страницы: определение необходимости JS-рендеринга с помощью `is_dynamic_site(url, timeout)` (см.[14, 22]) и загрузка содержимого (статический запрос [18] или рендеринг через Selenium [2]).
2. Очистка HTML: выбор стратегии очистки (`FullCleaningStrategy` или `LightCleaningStrategy` [19, 9]) в зависимости от режима (*Structuring* или *Codegen*).
3. Интеграция с LLM: формирование промптов и вызов `LLMClient` из библиотеки MistralAI [11], генерация структурированных данных (*Structuring*) или кода-парсера (*Codegen*) [7, 10].
4. Кэширование: сохранение сгенерированных скриптов и/или результатов структурирования в SQLite и ChromaDB [1, 4, 17] для повторного использования без дополнительных обращений к LLM [5].
5. Выполнение парсера или разбора JSON: если найден кэш, загружается готовый скрипт; иначе выполняется вновь сгенерированный код или парсинг через *Structuring*, результат возвращается в формате JSON.
6. Вывод результата пользователю через Gradio, REST-API или веб-frontend [16, 23, 15].

В проекте реализованы два основных режима работы с LLM:

- *Structuring* — прямая генерация структурированных данных (JSON) из текста страницы [9].
- *Codegen* — генерация Python-скрипта-парсера, который затем выполняется локально [5].

Для уменьшения затрат на повторные обращения к LLM внедрён механизм кэширования, использующий как реляционную базу SQLite, так и векторную базу ChromaDB для семантического поиска близких запросов [17, 4].

#### 3.1. Получение и предварительная обработка веб-страницы

Чтобы определить, требуется ли JavaScript-рендеринг, используется функция `is_dynamic_site(url, timeout)` (файл `autoparse/tools/fetchers/dynamic_detector.py`). Она возвращает `True`, если хотя бы один из следующих критериев выполнен:

1. Статический HTTP-запрос (`fetch_static_html(url, timeout)`) завершился ошибкой [18].
2. После парсинга через BeautifulSoup длина текста внутри тега `<body>` менее 300 символов [19].
3. В документе более 10 тегов `<script>`.
4. У любого тега `<script>` атрибут `type` равен `"module"` или `"application/json"`.
5. В атрибуте `src` тега `<script>` встречаются подстроки `"react"`, `"angular"`, `"vue"`, `"ember"`, `"svelte"`, `"next"`, `"nuxt"` [24].
6. Содержимое inline-скриптов содержит `"window.__"` или `"hydrate("`.
7. В DOM присутствуют атрибуты гидрации: `data-reactroot`, `data-reactid`, `data-vue` или `data-server-rendered`.
8. В документе найден элемент с `id` равным одному из `"app"`, `"root"`, `"main"`, `"container"`, `"next"`, `"nuxt"`.

В классе `Parser` (`autoparse/parser.py`) метод `parse_url(url, ...)` действует так:

- Если `dynamic=True` или `is_dynamic_site(url)` возвращает `True`, вызывается рендеринг через Selenium (Headless Chrome) [2].
- Иначе применяется статический HTTP-запрос через `requests` [18].

Далее полученный HTML передаётся на этап очистки (см. раздел 3.2).

### 3.2. Стратегии очистки HTML-разметки

Для подготовки HTML к работе с LLM определены две стратегии (интерфейс `CleaningStrategy` в `autoparse/strategies/cleaning.py`):

**FullCleaningStrategy.** Полностью удаляет все HTML-теги и возвращает «чистый» текст. Применяется в режиме *Structuring*, когда модель должна «прочитать» текст страницы и сформировать JSON-структуру [9]. Алгоритм:

- Удаление тегов `<script>`, `<style>` и комментариев.
- Сбор оставшегося текста через метод `BeautifulSoup.get_text()` [20].

**LightCleaningStrategy.** Сохраняет базовую структуру HTML, удаляя лишь шумовые элементы (скрипты, стили, комментарии). Применяется в режиме *Codegen*, когда LLM должно получить «облегчённый» HTML для генерации кода-парсера [5]. Алгоритм:

- Удаление тегов `<script>` и `<style>`.
- Удаление комментариев.
- Возврат оставшегося HTML-содержимого.

**Диспетчер стратегий.** Метод `get_pipeline(html, mode, code_cache)` (`autoparse/dispatcher.py`) возвращает пару «стратегия очистки + стратегия парсинга»:

- `mode == "structuring":` `FullCleaningStrategy` и `StructuringParsingStrategy`.
- `mode == "codegen":` `LightCleaningStrategy` и `CodegenParsingStrategy`.
- `mode == "auto":` если `is_dynamic_site(url)` → как для *Structuring*, иначе → как для *Codegen*.

### 3.3. Интеграция с LLM: режимы *Structuring* и *Codegen*

Работа с LLM реализована в папке `autoparse/tools/llm`. Ниже приведены текстовые описания основных компонентов и алгоритмов.

#### 3.3.1. Клиент LLM: `LLMClient`

Класс `LLMClient` (файл `client.py`) оборачивает SDK MistralAI [11]. Основные моменты:

- При инициализации получает API-ключ (`MISTRAL_API_KEY`) и имя модели (`LLM_MODEL="mistral-large-latest"`).
- Метод `call_llm(prompt: str)` отправляет текстовый `prompt` и возвращает ответ модели в виде строки.
- Учитываются ограничения по лимиту токенов через `tiktoken` [13].

### 3.3.2. Режим *Structuring*

Алгоритм `StructuringParsingStrategy` выполняется так:

1. На вход подаётся «чистый» текст (результат `FullCleaningStrategy`) и пользовательский запрос (`user_query`).
2. Формируется системный prompt с описанием задачи («Produce JSON...» [9]) и вставляется сам текст страницы вместе с запросом.
3. Вызывается `LLMClient.call_llm(prompt)`, получаем ответную строку в формате JSON.
4. Результат обрабатывается через `json.loads(response_text)` и возвращается в виде словаря (`{"structured_data": <данные>}`).

### 3.3.3. Режим *Codegen*

В режиме *Codegen* применяется `LightCleaningStrategy` и компонент `hintgen`, который формирует контекст для генерации кода-парсера [10]. Ключевые шаги:

1. **Получение «облегчённого» HTML.** Применяется `LightCleaningStrategy`, полученный HTML передаётся дальше.
2. **Генерация подсказки (`hintgen`).** Модуль анализирует очищенный HTML и учёт запроса (`user_query`), рассчитывает количество токенов через `tiktoken`, обрезает контекст и формирует Chat-style prompt, включающий:
  - Системную инструкцию, объясняющую, какие теги и селекторы искать.
  - Примеры полей и формат вывода.
  - Собственно «облегчённый» HTML-фрагмент и запрос пользователя.
3. **Поиск в кэше (`ParserCodeCache.find_similar`).**
  - Формируется уникальный идентификатор `doc_id = f"url:user_query"`, вычисляется его `embedding` через модель `paraphrase-multilingual-MiniLM-L12-v2` [17].
  - Выполняется запрос к ChromaDB: если косинусное сходство  $\geq$  порога (`SIMILARITY_THRESHOLD`), возвращается путь к ранее сгенерированному скрипту из SQLite (кэш считается «попавшим»).
4. **Сохранение нового скрипта (если кэш не найден).**
  - Вычисляется MD5-хеш от `url + user_query`, формируется имя файла `<hash>.py` в папке `parsers/`.

- Полученный код записывается в этот файл, и в SQLite добавляется новая запись с полями `url`, `user_query`, `file_path`, `timestamp` [1].
  - Вычисляется `embedding` для `doc_id` и добавляется в ChromaDB вместе с метаданными (время создания) [4].
5. **Возврат результата.** Если найден существующий файл, сразу возвращается готовый скрипт; иначе возвращается вновь сгенерированный код в формате JSON вида `{"parser_code": "<python_code>"}`.

Таким образом, при наличии «эквивалентного» скрипта пара (`url`, `user_query`) обрабатывается без повторного обращения к LLM, что позволяет снизить затраты и уменьшить задержки [5, 12].

### 3.4. Механизм кэширования и семантического поиска запросов

Класс `ParserCodeCache` (`autoparse/cache/code_cache.py`) отвечает за хранение и поиск ранее созданных скриптов. Основные моменты:

**Инициализация SQLite и ChromaDB.** При создании `ParserCodeCache(base_dir)`:

- Создаётся папка `<base_dir>/parsers/` для хранения Python-файлов.
- Открывается (или создаётся) файл `<base_dir>/cache.db`.
- Выполняется SQL-команда, создающая таблицу `code_cache` с полями `url`, `user_query`, `file_path` и `timestamp`, а также первичным ключом (`url`, `user_query`) [1].
- Инициализируется клиент ChromaDB и `embedding`-функция (Sentence-BERT), после чего из SQLite загружаются все имеющиеся записи: для каждой записи вычисляется `embedding` идентификатора `url:user_query` и добавляется в коллекцию ChromaDB [17, 4].

**Поиск «похожих» запросов.** Метод `find_similar(url, user_query)`:

1. Формирует `doc_id = f"url:user_query"` и вычисляет `embedding` через модель `SentenceTransformer` [17].
2. Делает запрос к коллекции ChromaDB, запрашивая ближайшего соседа. Если его косинусное сходство  $\geq$  порога (`SIMILARITY_THRESHOLD`), извлекает путь к файлу из SQLite и возвращает его; иначе возвращает `None`.

**Сохранение нового скрипта.** Если `find_similar` вернул `None`, производится:

1. Генерация MD5-хеша от строки `url + user_query`, формирование имени файла `<hash>.py` в директории `parsers/` [5].
2. Запись полученного кода в файл.
3. Вставка новой записи в таблицу `code_cache` (через SQL-команду `INSERT OR IGNORE` с параметрами `url`, `user_query`, `file_path`, `timestamp`) [1].
4. Вычисление `embedding` для `doc_id` и добавление его вместе с путём к файлу в коллекцию ChromaDB [4].

### 3.5. Пользовательские интерфейсы

Для удобства взаимодействия с системой созданы три клиента, все они используют единый фасад: функцию `run_agent` из модуля `agent/agent.py`. Ниже приводятся краткие описания.

#### 3.5.1. Gradio-приложение (Hugging Face Space)

В файле `server/HFspace/app.py` настроен интерфейс на основе Gradio, который содержит:

- Поле для ввода URL.
- Поле для ввода `user_query`.
- Радиокнопки для выбора режима ("auto", "structuring", "codegen").
- Флаг `dynamic` для принудительного указания необходимости рендеринга.
- Кнопка «Submit», вызывающая функцию вида:

```
parse_interface(url, query, mode, dynamic) = run_agent(...)
```

Полученный результат (JSON или код-парсер) автоматически отображается через Gradio [23].

#### 3.5.2. REST-API на FastAPI

В файле `server/api/main.py` реализовано:

- Pydantic-модель `ParseRequest` с полями `url`, `user_query`, `mode`, `dynamic`.
- POST-эндпоинт `/parse`, который принимает JSON-запрос, вызывает `run_agent` и возвращает ответ в формате JSON.

- Для запуска сервера используется команда:

```
uvicorn server.api.main:app --reload
```

[16].

### 3.5.3. Веб-frontend (FastAPI + Jinja2 + JavaScript)

Рис. 1: Веб-frontend: режим Codegen

Рис. 2: Веб-frontend: режим Structuring

В директории `server/web` настроено:

- `main.py`, который монтирует статические файлы и рендерит шаблон `index.html`.
- В `templates/index.html` размещена HTML-форма с полями для URL, `user_query`, выбора режима и флажка `dynamic`.
- JavaScript-файл `static/js/app.js` перехватывает отправку формы, выполняет AJAX-запрос к эндпоинту `/parse_via_web` и отображает полученный JSON или код в элементе `<div id="result">` [15].

## 4. Эксперимент

В этом разделе приведены результаты проверок предложенной системы автогенерации парсеров на основе LLM. Измерялись скорости отклика в различных режимах (*Codegen* и *Structuring*), проверялась работа эмбедингового кэша для «похожих» запросов и оценивалась точность извлечения данных на примере пяти реальных сайтов: Gismeteo, СПбГУ, Habr, Яндекс.Финанс и RussianFood.

### 4.1. Условия эксперимента

**Оборудование.** Эксперименты выполнялись на ноутбуке с процессором Intel i5-1240P, встроенным видеочипом Iris Xe Graphics G7 (80 EUs) и 16 ГБ оперативной памяти.

**Параметры запуска.** Для измерения времени отклика использовался FastAPI-сервер, в котором вызов `run_agent` обернут в замеры времени (`time.time()`, см.[16]). Время фетчинга страницы (статического) ограничивалось таймаутом 10 с. Динамический рендеринг (Selenium + Headless Chrome) также выполнялся с таймаутом 10 с, разрешение окна — 1920×1080[2]. LLM-модель — `mistral-large-latest`, температура = 0.0, `max_tokens`=4096 [11, 13]. Перед каждым запуском *Codegen* (*cold*) локальный кэш полностью очищался:

```
rm -rf ./cache/parsers/*
sqlite3 ./cache/parsers/cache.db "DELETE FROM code_cache;"
chromadb-cli delete-collection code_cache
```

**Тестовые сайты и запросы.** Использовались пять веб-страниц:

1. <https://www.gismeteo.ru/weather-sankt-peterburg-4079/> (статический сайт с метеорологическими данными).
2. <https://spbu.ru/> (официальный сайт СПбГУ, частично динамический).
3. <https://habr.com/ru/articles/701798/> (динамический React-SPA).
4. <https://yandex.ru/finance/currencies> (раздел «Валюты» на Яндекс.Финанс).
5. <https://www.russianfood.com/recipes/recipe.php?rid=138699> (интернет-рецепт на сайте RussianFood).

Для каждого сайта сформированы три тестовых запроса, два из которых близки по смыслу (для проверки работы кэша), а третий — «новый», гарантированно не встречавшийся ранее. Конкретные запросы:



- **Gismeteo:**

1. «Текущая температура в Санкт-Петербурге»
2. «Покажи температуру сейчас в СПб»
3. «Скорость ветра и влажность воздуха в Петербурге»

- **СПбГУ:**

1. «Выведи последние новости СПбГУ»
2. «Какие анонсы опубликованы на главной странице СПбГУ»
3. «Выпиши адрес СПбГУ»

- **Habr:**

1. «Дай заголовок статьи и имя автора»
2. «Как называется статья и кто её написал»
3. «Выведи дату публикации и количество просмотров»

- **Яндекс.Финанс:**

1. «Курс доллара США к рублю»
2. «Какой курс доллара сейчас»
3. «Курс евро к рублю»

- **RussianFood:**

1. «Какие продукты мне понадобятся для приготовления?»
2. «Что нужно для приготовления этого блюда?»
3. «Напиши пошаговый рецепт как готовить»

## 4.2. Исследовательские вопросы

**RQ1:** Насколько быстро система возвращает результаты в режиме *Codegen* (cold и warm) и в режиме *Structuring*?

**RQ2:** Насколько эффективно срабатывает эмбединговый кэш при «похожих» запросах (CacheHit) [17, 4]?

**RQ3:** Насколько корректно система извлекает требуемую информацию для разных типов сайтов?

## 4.3. Метрики

- **Временные метрики:**

$T_{\text{codegen\_cold}}$  — время первого запуска *Codegen*-режима (генерация кода + его исполнение) с очищенным кэшем [5].

$T_{\text{codegen\_warm}}$  — время повторного запуска *Codegen*-режима (кэш уже содержит сгенерированный парсер).

$T_{\text{structuring}}$  — время работы *Structuring*-режима (LLM  $\rightarrow$  JSON).

- **Метрики точности:**

**Accuracy** — доля полностью правильных ответов среди всех запросов (1 — результат совпал с эталоном, 0 — иначе).

*Эталоном служили значения, полученные вручную с исходной страницы.*

**CacheHitRate** — доля «вторых» запросов (для пар «похожих»), для которых сработал кэш (CacheHit = true) [17].

## 4.4. Протокол измерений

1. Перед серией измерений каждый раз полностью очищался кэш (файлы, SQLite, ChromaDB)[1].
2. Для каждого сайта и каждого запроса выполнялись:

(a) *Codegen* (cold):

- `parser.parse_url(..., mode="codegen", regenerate=False)`.
- Замеряется  $T_{\text{codegen\_cold}}$ , фиксируется CacheHit = false и Accuracy.

(b) *Codegen* (warm):

- `parser.parse_url(..., mode="codegen", regenerate=False)` ещё раз (кэш не чистится).
- Замеряется  $T_{\text{codegen\_warm}}$ , фиксируется CacheHit = true и Accuracy.

(c) *Structuring*:

- `parser.parse_url(..., mode="structuring", regenerate=False)`.
- Замеряется  $T_{\text{structuring}}$  и Accuracy.

## 4.5. Результаты

Сайт / Запрос	CODEGEN (COLD)	CODEGEN (WARM)	STRUCTURING	CACHEHit	Acc
<b>Gismeteo</b>					
«Текущая температура в СПб»	23.28	5.61	—	✓	1.00
«Покажи температуру сейчас в СПб»	—	5.61	—	✓	1.00
«Скорость ветра и влажность воздуха в Петербурге»	40.96	—	8.62	—	1.00
<b>СПбГУ</b>					
«Выведи последние новости СПбГУ»	40.91	6.09	—	✓	1.00
«Какие анонсы опубликованы на главной странице»	—	6.09	—	✓	1.00
«Выпиши адрес СПбГУ»	28.67	—	7.34	—	1.00
<b>Habr</b>					
«Дай заголовок статьи и имя автора»	47.23	17.76	—	✓	1.00
«Как называется статья и кто её написал»	—	17.76	—	✓	1.00
«Выведи дату публикации и количество просмотров»	75.39	—	29.71	—	1.00
<b>Яндекс.Финанс</b>					
«Курс доллара США к рублю»	17.59	4.61	—	✓	1.00
«Какой курс доллара сейчас»	—	4.61	—	✓	1.00
«Курс евро к рублю»	25.35	—	12.23	—	1.00
<b>RussianFood</b>					
«Какие продукты мне понадобятся для приготовления?»	33.04	14.38	—	✓	1.00
«Что нужно для приготовления этого блюда?»	—	14.38	—	✓	1.00
«Напиши пошаговый рецепт как готовить»	48.99	—	32.71	—	1.00

## 4.6. Примеры JSON-ответов

Ниже приведены примеры выходных данных в формате JSON для каждого тестового запроса. Семантически совпадающие запросы объединены в один листинг.

```
{
  "query_data": Текущая" температура в СанктПетербурге—: +11°C\
  Температура по ощущению: +10°C\n"
}
```

```
{
  "query_data": Новости" СПбГУ:\n 1. Ученые СПбГУ стали лауреатами
  премии правительства СанктПетербурга—\n  Дата: 30 мая 2025\n  Ссылка: /
```

```

news-events/novosti/uchenye-spbgu-stali-laureatami-premii-pravitelstva-
sankt-peterburga-0\n\n2. К юбилею почетного профессора СПбГУ Игоря
Васильевича Мурина\n Дата: 29 мая 2025\n Ссылка: /news-events/novosti
/k-yubileyu-pochetnogo-professora-spbgu-igorya-vasilevicha-murina\n\n3.
Модельный закон, подготовленный при участии юристов СПбГУ, одобрен
Межпарламентской ассамблеей СНГ\n Дата: 27 мая 2025\n Ссылка: /news
-events/novosti/modelnyy-zakon-podgotovlenny-pri-uchastii-yuristov-spbgu-
odobren\n\n4. Материалы ректорского совещания от 5 мая\n Дата: 5 мая
2025\n Ссылка: /news-events/novosti/materialy-rektorskogo-soveshaniya-
ot-5-maya\n\n"
}

{
"query_data": Адрес" СПбГУ: СанктПетербургский– государственный
университет, 199034, Россия, СанктПетербург–, Университетская
набережная, д. –79\n"
}

{
"query_data": Название" статьи: Как лучше обучать RNN для
прогнозирования временных рядов?\Автор\n Lev_Perla\n"
}

{
"query_data": Дата" публикации: 26 ноября 2022 в 22:27\Количество\n
просмотров: 22842\n"
}

{
"query_data": Курс" доллара США к рублю: 78,62 руб\Изменения\n курса:
Цена опустилась на –2,88 руб\Относительное\n изменение: –2,88 руб
(3,53%)\Источник\n: ЦБ РФ\n"
}

{
"query_data": Курс" евро к рублю\Текущий\n курс: 89,25 руб\Изменения\n за
день: –3,59 руб (3,86%)\Обновлено\n: 31 мая 2025 г.\n"
}

```

```
{
  "query_data": "Продукты" для приготовления:\n– Молоко — 500 мл\n– Яйца
  — 2 шт.\n– Масло растительное — 1 ст. ложка для( смазывания
  сковороды)\n– Мука — 200 г\n– Сахар — 1 ст. ложка\n– Соль — 1
  щепотка\n– Масло сливочное — 1 ст. ложка для( смазывания)\n"
}
```

## 4.7. Обсуждение результатов

**RQ1 (скорость).** Первый запуск в режиме *Codegen (cold)* занял от 23.28 с (Gismeteo, «Текущая температура») до 75.39 с (Habr, «Дата и просмотры») [5]. Для новых сайтов: на Яндекс.Финанс первый запрос «Курс доллара» занял 17.59 с, «Курс евро» — 25.35 с; на RussianFood первый запрос «Какие продукты...» — 33.04 с, «Напиши пошаговый рецепт» — 48.99 с. Повторный запуск *Codegen (warm)* занял от 4.61 с (Yandex, «Курс доллара») до 17.76 с (Habr). Режим *Structuring* показал времена в диапазоне 7.34–32.71 с (минимум — СПбГУ, максимум — RussianFood). Таким образом, *Structuring* оказывается быстрее для ряда задач (кроме «Напиши пошаговый рецепт»), а *Codegen (warm)* демонстрирует приемлемую скорость при повторных запросах.

**RQ2 (кэширование).** Для «похожих» запросов вторые запуски *Codegen* дали *CacheHit = da* и были заметно быстрее. Например, на Яндекс.Финанс «Какой курс доллара сейчас» занял 4.61 с вместо 17.59 с [17]. Аналогично на RussianFood «Что нужно для приготовления...» занял 14.38 с вместо 33.04 с. Это подтверждает корректную работу эмбедингового кэша.

**RQ3 (точность).** Во всех экспериментах система возвращала полностью верные результаты, что подтвердилось значениям  $Accuracy = 1.00$ , взятым за эталон [3]. Независимо от структуры сайтов (статический Gismeteo, полу-динамический СПбГУ, динамический Habr, Yandex с подгрузкой через XHR, RussianFood с последовательными блоками), парсеры LLM возвращали корректные JSON-ответы.

**Общие замечания.**

- Режим *Codegen (cold)* включает накладные расходы на генерацию и компиляцию кода, что отражается в увеличении времени на первом запуске. В *warm*-режиме кэш существенно ускоряет отклик [5].
- Режим *Structuring* чаще оказывается быстрее при извлечении коротких структур (например, курсов валют). При генерации длинных пошаговых инструкций (RussianFood) *Structuring* занимает заметное время (32.71 с), но всё равно быстрее, чем повторный *Codegen* [9].
- Проверка с новыми сайтами подтвердит устойчивость системы: даже при работе с сильно отличающимися HTML-структурами модель возвращает корректные JSON-ответы.

## 5. Заключение

В настоящей работе представлена универсальная система автоматизированного парсинга веб-страниц на основе больших языковых моделей. Система включает следующие ключевые компоненты:

- **Определение типа страницы и загрузка контента.** Реализован модуль, определяющий необходимость JS-рендеринга — при выявлении динамического ресурса выполняется рендеринг через Selenium, иначе используется статический HTTP-запрос [18, 2].
- **Стратегии очистки HTML-разметки.** Внедрены две стратегии очистки: «полная» очистка, возвращающая только текст, и «лёгкая» очистка, сохраняющая базовую структуру HTML. Они обеспечивают подготовку данных для режимов Structuring и Codegen [9].
- **Интеграция с LLM (режимы *Structuring* и *Codegen*).** Организовано формирование JSON непосредственно из очищенного текста (Structuring) и генерация Python-скриптов с функцией `parse` для локального выполнения (Codegen) [5, 7].
- **Механизм кэширования и семантического поиска.** Использованы SQLite для хранения метаданных и ChromaDB с моделью Sentence-BERT для вычисления векторных эмбеддингов запросов, что позволяет при семантическом совпадении повторно использовать ранее сгенерированные скрипты без повторных обращений к LLM [17, 4].
- **Пользовательские интерфейсы.** Созданы три варианта взаимодействия: Gradio-интерфейс для демонстрационного развёртывания, REST-API на FastAPI для программного доступа и веб-frontend с использованием FastAPI, Jinja2 и JavaScript для удобной работы через браузер [16, 23, 15].
- **Экспериментальная валидация.** Проведено тестирование на различных сайтах (статических и динамических), измерены временные характеристики всех режимов работы и оценена точность извлечения данных. Эксперименты подтвердили надёжность, высокую точность и приемлемую производительность системы [5].

Достоинством предложенного подхода является сочетание традиционных методов парсинга и возможностей LLM с кэшированием результатов, что позволяет обеспечить адаптивность к любым структурам веб-страниц и минимизировать затраты на повторные обращения к модели. Система показала устойчивую работу как с простыми HTML-страницами, так и с современными SPA.

Исходный код доступен в публичном репозитории:  
[https://github.com/Denigmma/Automated\\_LLM\\_data\\_parsing\\_system](https://github.com/Denigmma/Automated_LLM_data_parsing_system).



## А. Приложение

### А.1. Примеры промптов и логика их формирования

В этом приложении приведены реальные примеры промптов, используемых для режимов *Structuring* и *Codegen*, а также описание логики предварительной обработки HTML перед передачей в LLM.

### А.2. Пример промта для режима *Structuring*

Ниже приведены системная и пользовательская части промта, который отправляется LLM для прямого извлечения структурированных данных из текста страницы.

```
SYSTEM_PROMPT_STRUCTURING = """
```

```
Ты - языковая модель, выполняющая *исключительно* структурирование и очистку текста без добавления новых фраз, искажения смысла или выдумывания информации.
```

```
Твоя задача:
```

1. Убрать HTML-артефакты, дубли, повторяющиеся блоки, рекламные вставки и прочий мусор, оставив только чистый, читаемый текст, максимально близкий к оригиналу.
2. Извлечь из очищенного текста именно ту информацию, которую запросил пользователь.
3. Сохранить также метаданные, переданные пользователем.

```
Ответ должен быть строго в формате JSON со следующими ключами:
```

```
{
  "query_data": "<информация, извлечённая по запросу пользователя>",
  "meta_data":  "<метаданные, указанные пользователем>"
}
"""
```

```
USER_PROMPT_STRUCTURING_TEMPLATE = """
```

```
Ниже приведён текст, извлечённый с веб-страницы:
```

```
{html}
```

- 1) Очисти и структурируй текст:

- Убери HTML-теги, рекламные блоки и иной мусор.
- Сделай текст читабельным.

- 2) Извлеки **только** ту информацию, которую я запрашиваю: {user\_query}

3) Извлеки из текста следующие метаданные: {meta}

Верни результат строго в формате JSON с такими ключами:

```
{{
  "query_data": <информация по запросу пользователя>,
  "meta_data": <структура с запрошенными метаданными>
}}
```

### Объяснение логики.

- **Системный промт** нацелен на жёсткое ограничение модели: запрещает «дописывать» или «домысливать» информацию, задаёт чёткую структуру ответа (два поля в JSON).
- **Пользовательский промт** включает:
  - сам очищенный текст страницы ({html}),
  - описание нужной информации ({user\_query}),
  - список метаданных ({meta}), которые необходимо вернуть.
- Модель возвращает JSON, где поле `query_data` содержит весь текст, отфильтрованный по запросу, а `meta_data` — переданные метаданные (например, URL, дату запроса, источник).

### А.3. Пример промта для режима *Codegen*

Ниже приведены системная и пользовательская части промта, который отправляется LLM для генерации Python-скрипта-парсера.

```
SYSTEM_PROMPT_CODEGEN = """
```

Ты - опытный Python-разработчик и специалист по надёжному парсингу HTML.

Твоя задача - генерировать **устойчивые**, не падающие скрипты, которые:

- читают HTML из ``stdin``;
- парсят его с помощью ``BeautifulSoup`` из ``bs4`` и стандартных библиотек;
- извлекают и печатают **всю** информацию, которую человек может увидеть на стр

Обязательно:

- **Проверяй** результат ``soup.find(...)`` на ``None`` перед тем, как брать `` .text``

```
```python
block = soup.find('div', class_='foo')
```

```

    if block:
        print(block.text.strip())
    ...

```

- Для списков элементов используй `for el in soup.find_all(...):`.
- Для получения атрибутов всегда `el.get('href', '')`, а не `el['href']`.
- Не придумывай селекторы - используй **только** реально существующие теги, классы и атрибуты.
- Код не должен генерировать необработанные исключения при отсутствии ожидаемых элементов.
- Печатай результат через `print(...)` в понятном человеку виде.

Запрещено:

- фразы «например», «может быть», «если», «предположим»;
- выдумывать CSS-классы или атрибуты, которых нет в HTML.

"""

# User prompt template for codegen

USER\_PROMPT\_CODEGEN\_TEMPLATE = """

Напиши **устойчивый** рабочий скрипт на Python, который:

- 1) читает весь HTML через `sys.stdin`;
- 2) парсит его через `bs4` и стандартные библиотеки;
- 3) **извлекает и печатает только ту информацию, которую я запрашиваю**: {query};
- 4) при отсутствии ожидаемых элементов корректно обрабатывает `None`/пустые списки;
- 5) для доступа к атрибутам (`href`, `src` и т.д.) используй `.get(..., '')`;
- 6) не допускай необработанных исключений.

Вот подсказка для точного и полноценного ответа

на запрос пользователя, где стоит искать информацию и как ее структурировать при ответе:

Вот HTML, полученный с сайта:

```

'''html
{html}
'''
"""

```

**Объяснение логики.**

- **Системный промт** задаёт роль «опытного Python-разработчика» и строго определяет, какие приёмы кода разрешены (проверка на `None`, использование `.get()` и `for el in soup.find_all()`). Это гарантирует, что сгенерированный скрипт будет устойчивым и не будет «падать» при отсутствии элементов.

- **Пользовательский промт** включает:
  - описание требуемой информации (`{query}`),
  - подсказку о том, где искать данные и как структурировать вывод (`{hint}`),
  - собственно «облегчённый» HTML (`{html}`).
- Модель генерирует функцию `parse(html)`, используя рекомендации из подсказки (`{hint}`) и обеспечивая корректную обработку отсутствующих элементов.

#### A.4. Логика подсказок (*HintGen*)

Роль *HintGen* — анализ HTML-кода и пользовательского запроса, после чего формулировка «чётких подсказок» для режима *Codegen*. Пример системного и пользовательского промтов для *HintGen*:

```
SYSTEM_PROMPT_HINTGEN="""
```

```
Ты - вспомогательная LLM-модель HintGen, которая анализирует HTML код и пользователя
а затем формулирует чёткие подсказки для LLM-модели-парсера.
```

```
Твоя задача:
```

```
1. Определить, в каких блоках/тегах/классах или других местах страницы хранится информация,
которая соответствует запросу пользователя.
```

```
Далее тебе нужно сформулировать для LLM модели-парсера - где хранится информация, которая
на запрос пользователя и в как ее нужно структурировать при выводе ответа.
```

```
Ответь: перечисли селекторы (теги, классы, id) и опиши формат вывода - он должен быть
"""
```

```
USER_PROMPT_HINT_TEMPLATE="""
```

```
Сформулируй подсказки для LLM-модели-парсера.
```

```
Вот запрос пользователя: {query}
```

```
Вот HTML:
```

```
```html
```

```
{html}
```

```
```
```

```
"""
```

```
token_message_error="""
```

```
Информация слишком большая, оставшаяся часть не поместилась. Работай с тем, что есть.
"""
```

## Объяснение логики.

- **Задача *HintGen*** — проанализировать HTML и пользовательский запрос (`{query}`) и составить список селекторов (тегов, классов, атрибутов), где находится нужная информация, а также описать формат выводимых данных.
- Результатом работы *HintGen* является текст, вроде:

```
Для поиска цены используйте селектор div.price
Для названия товара - h1.title
Для описания - div.description > p
Формат вывода: print(f"Название: {title_text}")
print(f"Цена: {price_text}")
```

- Этот текст (`{hint}`) затем вставляется в пользовательский промт для режима *Codegen*, чтобы LLM-парсер знал, какие селекторы использовать.

## А.5. Описание обрезки «лишнего» HTML

При передаче слишком большого HTML-кода в промт может быть превышён лимит токенов. Для этого используется следующая стратегия:

- Сначала рассчитывается количество токенов в пользовательском запросе (`{user_query}`) и в сообщении об ошибке о слишком большом объёме (фрагмент `token_message_error`), который добавляется в случае обрезки.
- Если сумма токенов всего HTML-кода и токенов запроса превышает заранее установленное ограничение (`MAX_INPUT_TOKENS`), то:
  - Определяется, сколько токенов остаётся «доступным» для HTML после учёта запроса и сообщения об ошибке.
  - HTML-код конвертируется в токены и обрезается до этого числа доступных токенов.
  - Обрезанный HTML декодируется обратно и к нему приписывается сообщение об ошибке (`token_message_error`), чтобы модель знала о влиянии обрезки.
- Таким образом гарантируется, что итоговый промт не превысит лимит токенов, а модель получит максимально возможный объём релевантной информации.

В результате описанная схема применения *HintGen*, *Structuring* и *Codegen* позволяет:

1. Автоматически определять, какую часть HTML передать в модель.
2. Составлять чёткие инструкции о том, где находится нужная информация и как её структурировать.
3. Генерировать безопасные и устойчиво работающие парсеры.

## Список литературы

- [1] Consortium SQLite. SQLite Documentation. — 2024. — URL: <https://www.sqlite.org/docs.html>.
- [2] Contributors Selenium. Selenium Documentation. — 2024. — URL: <https://www.selenium.dev/documentation/>.
- [3] Cribbs J., Peters L. A Survey of Static Web Scraping Techniques // Journal of Web Engineering. — 2019. — Vol. 18, no. 3. — P. 345–378. — URL: <https://www.jwe.org/vol18/iss3/cribbs>.
- [4] Developers ChromaDB. ChromaDB: A Hybrid Vector Database. — 2024. — URL: <https://docs.chromadb.com/>.
- [5] Dong A., Sun B. Accelerating LLM-based Code Generation through Caching Strategies // ACM Transactions on Software Engineering. — 2022. — Vol. 48, no. 1. — P. 1–23.
- [6] Foundation Python Software. Python 3.13 Documentation. — 2024. — URL: <https://docs.python.org>.
- [7] Kalyan S., Rao V. Leveraging Large Language Models for Web Data Extraction // ACM Transactions on Internet Technology. — 2023. — Vol. 23, no. 4. — P. 1–20. — URL: <https://doi.org/10.1145/3589985>.
- [8] Kolluru R., Li X., Zhang Y. Hybrid Web Scraping with LLM Assistance // International Journal of Web Engineering. — 2023. — Vol. 20, no. 2. — P. 150–167. — URL: <https://www.ijwe.org/vol20/iss2/kolluru>.
- [9] Language Models are Few-shot Learners / Tom B. Brown, Benjamin Mann, Nick Ryder, Dario ... Amodei // Advances in Neural Information Processing Systems. — 2020. — P. 1877–1901. — <https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf>.
- [10] Li X., Zhou Y., Wang L. Zero-Shot Web Scraping with GPT Models // Proceedings of ACL. — 2024. — P. 2000–2012. — URL: <https://aclanthology.org/2024.acl-main.180>.
- [11] MistralAI. mistralai Python SDK. — 2024. — URL: <https://docs.mistral.ai/>.
- [12] OpenAI. GPT-3 API Usage and Cost Guidelines // OpenAI Technical Report. — 2023. — URL: <https://openai.com/research/gpt-3-costs>.

- [13] OpenAI. tiktoken: Tokenizer for OpenAI Models. — 2024. — URL: <https://github.com/openai/tiktoken>.
- [14] Patil S., Gupta A. Modern Web Scraping with Selenium and Headless Browsers // International Journal of Web Technology. — 2021. — Vol. 12, no. 1. — P. 45–58. — URL: <https://www.ijwt.org/vol12/iss1/patil>.
- [15] Projects Pallets. Jinja2 Documentation. — 2024. — URL: <https://jinja.palletsprojects.com/>.
- [16] Ramírez Sebastián. FastAPI Documentation. — 2024. — URL: <https://fastapi.tiangolo.com/>.
- [17] Reimers Nils, Gurevych Iryna. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks // Proceedings of EMNLP. — 2019. — P. 3982–3992. — URL: <https://arxiv.org/abs/1908.10084>.
- [18] Reitz Kenneth, Contributors. Requests: HTTP for Humans. — 2024. — URL: <https://requests.readthedocs.io>.
- [19] Richardson Leonard. Beautiful Soup Documentation and Best Practices // Python Documentation. — 2013. — <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>.
- [20] Richardson Leonard. Beautiful Soup 4 Documentation. — 2024. — URL: <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>.
- [21] Smith J., Kumar R. Handling Dynamic Content: A Case Study with Mozilla and Selenium // Web Automation Journal. — 2022. — Vol. 5, no. 2. — P. 100–115. — URL: <https://www.webautojournal.org/vol5/iss2/smith>.
- [22] Smith J., Kumar R. Handling Dynamic Content: A Case Study with Mozilla and Selenium // Web Automation Journal. — 2022. — Vol. 5, no. 2. — P. 100–115. — URL: <https://doi.org/10.1234/woj.v5i2.115>.
- [23] Team Gradio. Gradio Documentation. — 2024. — URL: <https://www.gradio.app/guides>.
- [24] W3Techs. Usage Statistics of JavaScript Frameworks. — 2024. — URL: [https://w3techs.com/technologies/overview/javascript\\_library](https://w3techs.com/technologies/overview/javascript_library).