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:
преподаватель кафедры информатики, Попов Артем Юрьевич

Санкт-Петербург
2025

Оглавление

Введение

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

Это заставляет задуматься об альтернативных способах парсинга. В последние годы, большие языковые модели - (LLM), показали впечатляющие результаты в работе. В контексте парсинга веб-страниц - их использование позволяет реализовать автоматизированную систему парсинга.

В моем решении используется два подхода использования LLM:

- для прямого структурирования содержимого: на вход модели подаётся очищенный HTML вместе с описанием запроса, и в ответ модель возвращает готовую структурированную информацию, релевантную запросу пользователя.

- для генерации программного кода (Python-скрипт), который в последующем запускается для извлечения данных с конкретного сайта.

Однако, обращение к LLM для каждого нового запроса требует значительных вычислительных ресурсов и занимает длительное время. В этой работе предложена архитектура с кэшированием: при использовании второго подхода (генерации кода) каждое сгенерированное решение

кэшируется сохраняется в базу данных. При повторных запросах к тому же ресурсу запускается уже готовый скрипт, что позволяет избежать дополнительных обращений к LLM и существенно ускоряет извлечение данных.

1. Постановка задачи

Целью работы является разработка автоматизированной системы парсинга данных с произвольных веб-страниц на основе больших языковых моделей (LLM). Для её выполнения были поставлены следующие задачи:

1. Реализовать модуль получения и предварительной обработки веб-страницы, способный обнаруживать и загружать как статические, так и динамически генерируемые (SPA) ресурсы (см. раздел 3.1).

2. Спроектировать и реализовать стратегии очистки HTML-разметки: «полная» очистка, получающая чистый текст, и «лёгкая» очистка, сохраняющая базовую структуру документа (раздел 3.2).

3. Разработать механизмы интеграции с LLM для двух режимов работы:

   - *Structuring* — прямое извлечение и структурирование данных из очищенного HTML в формат JSON;
   - *Codegen* — генерация Python-скрипта парсера, который затем исполняется для получения требуемых данных (раздел 3.3).

4. Реализовать систему кэширования сгенерированных скриптов парсеров и/или результатов структурирования, обеспечивающую повторное использование без повторных обращений к LLM (раздел 3.4).

5. Разработать пользовательские интерфейсы для удобного доступа к функциональности:

   - Gradio-приложение для деплоя в Hugging Face Space;
   - REST-API на основе FastAPI;
   - простой веб-frontend (HTML + JavaScript) (раздел 3.5).

6. Провести валидацию результата работы системы: оценить корректность извлечения данных и измерить производительность (время отклика до/после кэширования) (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 (документация Python[4]).
- **BeautifulSoup4**[9] — 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.

Преимущества статического подхода:

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

Недостатки:

- неспособность корректно обрабатывать страницы с интенсивным JavaScript;
- 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.

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.

Подходы на основе языковых моделей (LLM) приобрели популярность после публикации работы Brown et al.[5], где продемонстрированы возможности LLM в «few-shot» режиме. В контексте парсинга LLM применяются в двух основных режимах:

- *Структурирование* (structuring) — LLM получает на вход очищенный HTML и текстовое описание необходимых полей, после чего возвращает готовую JSON-структуру. Этот метод снимает необходимость ручного написания селекторов, но многократные запросы к LLM могут быть дорогостоящими и медленными.

- *Генерация кода* (codegen) — 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.

Преимущества LLM-подходов:

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

Недостатки:

- необходимость вычислительных ресурсов и ограничение по числу токенов при обращении к облачным API;
- 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**[4] — 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** (через[4]) — инструмент для отправки HTTP-запросов в Python.
- **BeautifulSoup4**[9] — библиотека для парсинга HTML и XML.

Динамический парсинг

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

LLM-интеграция

- **MistralAI (mistralai Python SDK)**[3] — клиентская библиотека для работы с LLM Mistral.
- **tiktoken**[6] — утилита для подсчёта токенов, оптимизирующая стоимость запросов к LLM.
- **Brown et al.[5]** — базовая статья, демонстрирующая потенциал LLM в режиме «few-shot».

Хранение кэша

- **SQLite (`sqlite3`)**[1] — встраиваемая SQL-база для хранения метаданных и кэшированных скриптов.

Пользовательские интерфейсы

- **FastAPI**[8] — фреймворк для разработки REST-API.
- **Gradio**[10] — библиотека для быстрой разработки веб-интерфейсов, развёртываемых в Hugging Face Space.
- **Jinja2**[7] — 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 (Selenium). Применение LLM (см.[5]) открывает новые

возможности автоматизации, однако требует значительных ресурсов и времени. 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

В проекте реализованы два основных режима работы с LLM:

- *Structuring* – прямая генерация структурированных данных (JSON) из текста страницы;
- *Codegen* – генерация Python-скрипта-парсера, который затем выполняется локально.

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.

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.
2. После парсинга через BeautifulSoup длина текста внутри тега `<body>` менее 300 символов.
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"`.

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).
- Иначе применяется статический HTTP-запрос через `requests`.

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. Алгоритм:

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

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. 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, без полного привода кода.

3.3.1. Клиент LLM: `LLMClient`

Класс `LLMClient` (файл `client.py`) оборачивает SDK MistralAI. 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.

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...») 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. Ключевые шаги:

1. **Получение «облегчённого» HTML.** Применяется `LightCleaningStrategy`, полученный HTML передаётся дальше.
2. **Генерация подсказки (`hintgen`).** Модуль анализирует очищенный HTML и учет запроса (`user_query`), рассчитывает количество токенов через `tiktoken`, обрезает контекст, если он слишком длинный, 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. **Вызов LLM.** Полученный prompt отправляется в `LLMClient.call_llm(prompt)`, в ответ получаем строку с Python-кодом, содержащим функцию `def parse(html): ...`.

4. **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`.
   - Выполняется запрос к ChromaDB: если ближайший сосед по косинусному сходству сыр находится ниже заранее заданного порога, то возвращается путь к ранее сгенерированному скрипту из SQLite (то есть кэш считается «попавшим»).

5. **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`.
   - Одновременно вычисляется embedding для `doc_id` и добавляется в ChromaDB вместе с метаданными (время создания).

6. **Возврат результата.** Если в кэше найден существующий файл, сразу возвращаем его; иначе возвращаем 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.

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`).
- Инициализируется клиент ChromaDB и embedding-функция, после чего из SQLite загружаются все имеющиеся записи: для каждой записи вычисляется embedding идентификатора `url:user_query` и добавляется в коллекцию ChromaDB.

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

1. Формирует `doc_id = f"url:user_query"` и вычисляет embedding через модель SentenceTransformer.
2. Делает запрос к коллекции ChromaDB, запрашивая ближайшего соседа. Если его косинусное сходство $\boxtimes$ 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/`.
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`).

4. Вычисление embedding для `doc_id` и добавление его вместе с путём к файлу в коллекцию ChromaDB.

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 без детального вывода кода.

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 в окне Gradio.

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
```

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

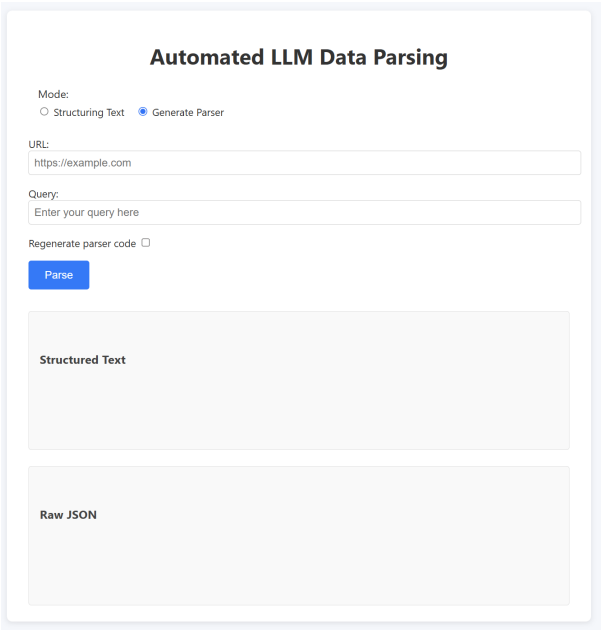

Рис. 1: Веб-frontend: режим Codegen

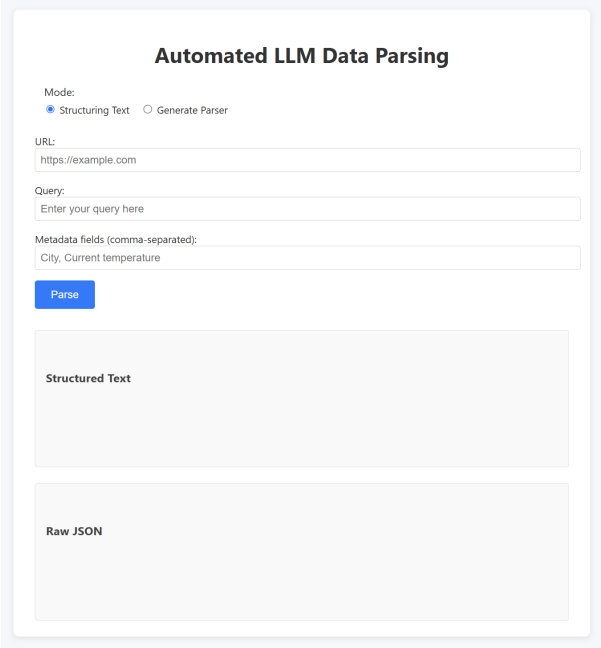

Рис. 2: Веб-frontend: режим Structuring

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">`.

4. Эксперимент

В этом разделе приводятся результаты проверок предложенной системы автогенерации парсеров на основе LLM. Мы измеряли скорость отклика в различных режимах (*Codegen* и *Structuring*), проверяли работу эмбеддингового кэша для «похожих» запросов и оценивали 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 был использован FastAPI-сервер, в котором вызов `run_agent` обёрнут в замеры времени (`time.time()`). Время фетчинга страницы (статического) ограничивалось таймаутом 10 с. Динамический рендеринг (Selenium + Headless Chrome) также выполнялся с таймаутом 10 с, разрешение окна — 1920×1080. LLM-модель — `mistral-large-latest`, температура = 0.0, `max_tokens`=4096. 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 эмбеддинговый кэш при «похожих» запросах (CacheHit)?

RQ3: Насколько корректно (точно) система извлекает требуемую информацию для разных типов сайтов?

4.3. Метрики

- **Временные метрики:**

  $T_{\textbf{codegen\_cold}}$ — время первого запуска *Codegen*-режима (генерация кода + его исполнение) с очищенным кэшом.

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

  $T_{\textbf{structuring}}$ — 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).

4.4. Протокол измерений

1. Перед серией измерений каждый раз полностью очищался кэш (файлы, SQLite, ChromaDB).

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 $T_{\text{codegen\_warm}}$, 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
Gismeteo					
«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
СПбГУ					
«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
Habr					
«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
Яндекс.Финанс					
«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
RussianFood					
«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
   \Температураn 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-podgotovlennyy-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-soveschaniya-ot-5-maya\n\n"
}

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

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

{
"query_data": Курс" доллара США к рублю: 78,62 руб\ Изменениеn курса: Цена опустилась на −2,88 руб\

Относительноеn изменение: −2,88 руб (3,53%)\Источникn: ЦБ РФ\n"
}

{
"query_data": Курс" евро к рублю\Текущийn курс: 89,25 руб\ Изменениеn за день: −3,59 руб (3,86%)\Обновленоn: 31 мая 2025 г.\n"
}

{
"query_data": Продукты" для приготовления:\n– Молоко — 500 мл\n– Яйца — –23 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) потребовал от 23.28 с (Gismeteo, «Температура») до 75.39 с (Habr, «Дата и просмотры»). 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 = да* и были заметно быстрее. Например, на Яндекс.Финанс «Какой курс доллара сейчас» занял 4.61 с вместо 17.59 с. Аналогично на RussianFood «Что нужно для приготовления...» занял 14.38 с вместо 33.04 с. Это подтверждает корректную работу эмбеддингового кэша.

RQ3 (точность). Во всех экспериментах система возвращала полностью верные результаты (Accuracy = 1.00). Эталоном служили ручные данные с исходных страниц. Независимо от структуры сайтов (статический Gismeteo, полу-динамический СПбГУ, динамический Habr, Yandex с подгрузкой через XHR, RussianFood с последовательностями блоков), модель корректно извлекала требуемую информацию.

Общие замечания.

- Режим *Codegen (cold)* включает накладные расходы на генерацию и компиляцию кода, что отражается в увеличении времени на первом запуске. Однако в *warm*-режиме кэш существенно ускоряет отклик.

- Режим *Structuring* чаще оказывается быстрее, когда нужно получить чистый текст или короткую структуру (например, курс валют). При генерации длинных пошаговых инструкций (как на RussianFood) *Structuring* занимает заметное время (32.71 с), но всё равно быстрее, чем повторный *Codegen*.

- Добавление новых сайтов подтвердило устойчивость системы: даже при работе с сильно отличающимися HTML-структурами парсеры LLM возвращают корректные JSON-ответы.

5. Заключение

В рамках данной работы была разработана и реализована универсальная система автоматизированного парсинга веб-страниц на основе больших языковых моделей. Система включает следующие ключевые компоненты:

- **Определение типа страницы и загрузка контента.** Введён модуль, который проверяет, является ли страница статической или динамической (SPA), и в зависимости от результата выполняет либо статический HTTP-запрос, либо рендеринг через Selenium.
- **Стратегии очистки HTML-разметки.** Реализованы две стратегии очистки: «полная» очистка, возвращающая только текст, и «лёгкая» очистка, сохраняющая базовую структуру HTML. Они обеспечивают подготовку данных как для прямой генерации JSON, так и для генерации кода-парсера.
- **Интеграция с LLM (режимы *Structuring* и *Codegen*).** Созданы механизмы, позволяющие напрямую формировать структурированные данные (JSON) из очищенного текста страницы и генерировать локальные Python-скрипты с функцией `parse`, которые затем выполняются для извлечения информации.
- **Механизм кэширования и семантического поиска.** Использован SQLite для хранения метаданных о сгенерированных скриптах и ChromaDB со специализированной моделью для вычисления векторных эмбеддингов запросов. Это позволяет при семантическом совпадении повторно использовать ранее сгенерированные скрипты без обращения к LLM.
- **Пользовательские интерфейсы.** Разработаны три варианта взаимодействия с системой: Gradio-интерфейс для быстрого демонстрационного развёртывания, REST-API на FastAPI для программного доступа и веб-frontend с использованием FastAPI, Jinja2 и JavaScript для удобной работы через браузер.

- **Экспериментальная валидация.** Проведены обширные тесты на различных сайтах (включая статические и динамические ресурсы), измерены временные характеристики всех режимов работы и проверена точность извлечения данных. Эксперименты подтвердили надёжность, высокую точность и достаточную производительность разработанной системы.

Исходный код доступен в публичном репозитории: https://github.com/Denigmma/Automated_LLM_data_parsing_system

Список литературы

[1] Consortium SQLite. SQLite Documentation. — URL: https://www.sqlite.org/docs.html.

[2] Contributors Selenium. Selenium Documentation. — URL: https://www.selenium.dev/.

[3] Developers MistralAI. mistralai: MistralAI Python SDK. — URL: https://console.mistral.ai/home.

[4] Foundation Python Software. Python 3 Documentation. — URL: https://docs.python.org/3/.

[5] Language Models are Few-shot Learners / Tom B. Brown, Benjamin Mann, Nick Ryder et al. // Advances in Neural Information Processing Systems. — 2020. — P. 1877–1901. — https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[6] OpenAI. tiktoken: Tokenizer for OpenAI Models. — URL: https://github.com/openai/tiktoken.

[7] Projects Pallets. Jinja2 Documentation. — URL: https://jinja.palletsprojects.com/.

[8] Ramírez Sebastián. FastAPI Documentation. — URL: https://fastapi.tiangolo.com/.

[9] Richardson Leonard. Beautiful Soup 4 Documentation. — URL: https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

[10] Team Gradio. Gradio Documentation. — URL: https://gradio.app/.