ChatGPT

Проектирование сервиса обработки документов (Unstructured + Donut + LLM)

Требования к выходным данным

Сервис должен принимать **разнообразные документы** (PDF, таблицы, изображения, сканы, фото экранов приложений, PDF с фото паспорта и рукописным адресом и т.д.) и возвращать **структурированный JSON** со следующими компонентами:

- **Описание документа естественным языком:** краткое человеко-понятное описание типа и содержимого документа.
- **Полный текст документа + описания визуальных элементов:** сплошной текст всего документа, включая *детализированные описания всех нетекстовых элементов* (изображений, штампов, схем и т.п.).
- **Чанки текста для RAG:** разбиение текста на фрагменты (чанки) подходящего размера для эмбеддинга, чтобы затем использовать их в retrieval-augmented generation.
- **Набор вопросов-ответов по каждой странице:** для каждой страницы документа – список из ~20–50 пар вопрос–ответ, сгенерированных на основе содержания страницы (с учётом всего контекста документа), отражающих ключевую информацию.
- **Исходные блоки документа:** коллекция всех выделенных блоков оригинального документа (текстовые блоки и изображения) – с их исходным содержимым (текстом блока либо извлечённым изображением).
- **Статистика по ресурсам:** данные о затратах ресурсов на разных этапах обработки – например, время выполнения каждого шага, загрузка CPU/GPU, количество токенов, потраченных каждой моделью, и пр. (для мониторинга производительности и масштабирования).

Общая архитектура системы

Решение строится как **многоступенчатый конвейер** обработки, где каждый этап дополняет предыдущий. В основе – комбинация библиотеки **Unstructured** для структурирования документов и моделей **Donut** и **LLM** для интеллектуального извлечения данных. Сервис будет развернут в корпоративной среде как Docker-контейнер с доступом по REST API (без авторизации, так как сервис используется внутри защищённого контура).

Основные компоненты архитектуры:

- **REST API сервис** (например, на FastAPI/Flask): принимает файлы/данные, управляет очередью заданий, возвращает статус и результаты.
- **Модуль парсинга документов**: использует библиотеку *Unstructured* для конвертации сырых файлов в структурированные элементы (текстовые абзацы, заголовки, изображения, таблицы

и т.д.) [1]. Этот модуль включает OCR для изображений/сканов и при необходимости задействует модель Donut для улучшения распознавания.
- **Модуль анализа контента (LLM)**: большая языковая модель (например, из семейства Qwen ~7–12B параметров) обрабатывает извлечённый текст, чтобы сгенерировать описание документа, вопросы-ответы и т.д. Также при необходимости подключается мультимодальная модель или модель описания изображений для нетекстовых элементов.
- **Хранилище временных данных**: локальная директория (или база) для сохранения загруженных файлов и промежуточных артефактов *с последующим удалением*. Файлы хранятся под случайными уникальными именами (для маскировки) и автоматически удаляются после обработки (не позднее, чем через N дней, по умолчанию 3 дня, если не удалены раньше).
- **Очередь заданий**: механизмы для очередной обработки документов. Поступающие файлы могут ставиться в очередь, чтобы не перегружать систему; при достаточных ресурсах возможна параллельная обработка нескольких документов одновременно.
- **Аппаратное ускорение**: GPU (40 ГБ) используется для ресурсоёмких задач – инференса моделей Donut и LLM. CPU занимается менее тяжёлыми задачами (например, парсинг PDF, запуск OCR), выполняя их параллельно с GPU-задачами когда возможно.

Ниже описаны ключевые этапы конвейера и решения, принятые при проектировании каждого из них.

Этап 1: Загрузка файла и управление очередью

Сервис предоставляет HTTP REST API для загрузки документов и получения результатов. Для надёжной передачи больших файлов реализована поддержка *chunked upload* (поштучная загрузка кусками):

1. **Инициализация загрузки:** клиент посылает запрос на начало загрузки; сервис возвращает уникальный *ключ загрузки* (идентификатор сессии).
2. **Передача чанков:** клиент разбивает файл на части $\leq$50 МБ и отправляет их последовательными запросами, указывая ключ загрузки и порядковый номер чанка. Сервис принимает куски и временно сохраняет их.
3. **Завершение загрузки:** клиент посылает финальный запрос с ключом загрузки, сигнализируя об окончании передачи. Сервис проверяет целостность – все ли чанки получены – и собирает их в исходный файл. В ответ возвращается *идентификатор загруженного файла* и его размер, либо ошибка, если какие-то части отсутствуют.

Для относительно небольших файлов предусмотрен упрощённый путь – однократная загрузка всего файла одним запросом (с мгновенным получением идентификатора файла в ответе).

После успешной загрузки документ ставится в очередь на обработку. Клиент может опрашивать статус:

- **GET /status/{file_id}:** возвращает состояние обработки по идентификатору (напр. "в очереди", "обрабатывается", "готово") и, если готово, ссылку на результат.
- **GET /result/{file_id}:** возвращает готовый JSON с результатами обработки (описание, текст, вопросы, и т.д.).

- **DELETE /file/{file_id}:** явное удаление всех хранимых данных по данному файлу (исходный файл, промежуточные артефакты, результат). Без явного запроса сервис автоматически очистит данные не позднее чем через 3 дня хранения.

Хранение файлов. Загруженный файл сохраняется на диск (например, в каталог `/data/uploads`) под UUID-именем. Это снижает риск коллизий и не позволяет по имени угадывать содержимое. При желании, файл может сохраняться в зашифрованном виде на диске или на уровне ФС могут быть права доступа только у сервиса. Пока файл ожидает обработки или обрабатывается, он доступен сервису. После формирования финального JSON файл может быть удалён немедленно (если он не нужен для отложенных задач) либо сохранён до истечения TTL в 3 дня (на случай повторного формирования результата или отладочных целей).

Очередь и параллелизм. Если одновременно загружается несколько документов, они ставятся в очередь. Сервис может обрабатывать их последовательно или параллельно, в зависимости от ресурсов и настроек. Поскольку 12B-модели занимают существенную память, разумно ограничить степень параллелизма – например, обрабатывать по одному документу за раз на GPU, либо запускать два потока, если каждая модель может быть частично выгружена/загружена. Мы можем реализовать *worker*-процесс, который берёт задания из очереди и выполняет конвейер обработки, чтобы основному веб-сервису не блокировать выполнение. Такой подход повышает надёжность: веб-слой быстро отвечает (200 OK статус "в очередь"), а тяжёлая обработка идёт в фоновой задаче.

Этап 2: Парсинг документа (Unstructured + OCR + Donut)

На этом этапе система конвертирует сырой файл в структурированное представление – **элементы документа**. Мы используем библиотеку **Unstructured**, которая предоставляет готовые компоненты для разбора различных форматов (PDF, DOCX, изображения, HTML и т.д.) и преобразования их в унифицированные объекты (типы элементов: заголовок, абзац, список, таблица, изображение и др.) [1]. Основные шаги:

1. Извлечение текста и разметки. Если входной PDF содержит текстовый слой (не отсканированный), Unstructured сможет напрямую извлечь текст без OCR. Сохраняется разбивка по абзацам, заголовкам, спискам и т.п., что упрощает последующую обработку. Если документ – это изображение или скан (например, фотография страницы, скан паспорта), применяется OCR. Мы настраиваем Unstructured использовать оптимальный OCR-движок: например, **PaddleOCR** вместо стандартного Tesseract для лучшей многоязычной поддержки [2]. PaddleOCR способен распознавать текст на разных языках (включая кириллицу, иероглифы и др.) и часто точнее Tesseract на сложных шрифтах. *(Unstructured позволяет задать OCR-агент через переменную окружения* `OCR_AGENT`*, поддерживая Tesseract, Paddle или Google Vision* [2]*.)*

В результате парсинга мы получаем список элементов с их метаданными: текстовые блоки (с самим текстом, языком, номером страницы, координатами на странице), изображений (с двоичными данными или ссылкой на файл изображения, координатами, размером), таблиц (можно получить как HTML-разметку таблицы) и пр. Таблицы по умолчанию могут быть конвертированы в текст (например, CSV через табуляцию), но чтобы **сохранить структуру таблиц**, мы можем воспользоваться возможностью Unstructured представлять таблицу в виде HTML в метаданных элемента [3]. Это сохранит строки/столбцы, что полезно для корректного смыслового анализа и генерации вопросов по таблицам.

2. Описание визуальных блоков. Для всех нетекстовых элементов – изображений, графиков, штампов, скриншотов – проводим *визуальный анализ*. Цель – встроить в итоговый текст документа человекочитаемое описание этих элементов. Здесь есть два подхода, которые можно комбинировать: - Использовать отдельную модель **image captioning** (например, BLIP-2 или подобную), которая по картинке генерирует описание на естественном языке. - Использовать мультимодальную LLM, такую как **Qwen-VL** (Vision-Language). Модель Qwen2-VL-7B способна воспринимать изображения и текст и показывала **состояние искусства в понимании документов** (например, на бенчмарке DocVQA у неё точность ~94.5% [4]). Она умеет извлекать текст с изображения и понимать сцены на разных языках [5]. Мы можем подать ей изображение блока и запрос типа: "Опиши изображение и текст на нём" – на выходе получить детальное описание. Такой подход избавляет от отдельного OCR для текста внутри изображения, так как модель сразу прочтёт и опишет, *что изображено на печати или фото* (например: "Фотография: цветной портрет, человек держит паспорт; на паспорте видны надписи..."). Однако Qwen-VL – тяжёлая модель (~7 млрд параметров + визуальный энкодер), поэтому можно применять её избирательно (например, только если простой OCR не справился или нужен действительно семантически осмысленный комментарий). В простых случаях может хватить и стандартного captioning.

3. Применение модели Donut для структурированных данных. Donut (Document Understanding Transformer) – это трансформер, который **прямо по изображению документа генерирует структурированный вывод** без отдельного OCR шага [6]. Его можно использовать, чтобы дополнить результаты стандартного парсинга в случаях, где нужен выделенный набор полей. Например, для **счётов, чеков, накладных** – Donut, обученный на таких документах, может сразу извлечь ключевые поля (номер счёта, дата, сумма, адрес и т.д.) и отдать JSON с этими полями [7]. Мы можем интегрировать Donut следующим образом: - Если при разборе документа мы распознали определённый *тип* (скажем, по наличию слова "Invoice" или формату – таблицы сумм), то дополнительно прогоняем изображение страницы через соответствующую модель Donut, настроенную на этот тип документа. - Donut-вывод (JSON со структурой полей) затем включаем в результирующий JSON (например, вложенным объектом "extracted_fields" для этой страницы). Это обогатит результат конкретными структурированными данными. **Unstructured+Donut подход** как раз разрабатывается командой Unstructured для обработки квитанций и счетов [8], что подтверждает эффективность комбинации: классические методы дают общую структуру и текст, а Donut точечно выцепляет нужные данные. - Даже если тип документа неизвестен, Donut может быть полезен как альтернативный способ OCR: он **OCR-free** – то есть сам обучен распознавать символы и макет, поэтому может быть устойчивее к нестандартным шрифтам, языкам или низкому качеству скана [6]. В сложных случаях можно запустить Donut в режиме "прочти всё, что видишь" (с соответствующим промптом) и сравнить с результатом OCR. Комбинируя оба, мы повышаем точность: расхождения можно логически слить или выдать оба варианта в «исходниках блоков». Например, если Tesseract не уверен в слове, Donut может правильно распознать его контекстно.

4. Формирование структуры по страницам. После обработки у нас есть множество элементарных блоков. Мы сгруппируем их по номерам страниц, сохраняя порядок следования. В каждом блоке: - Текстовые блоки – уже содержат извлечённый текст. - Изображения – заменяем (или дополняем) *описанием*, полученным от caption-модели/LLM, чтобы впоследствии включить его в общий текст. Сами двоичные данные изображения можем сохранить отдельно (например, временно файл или закодировать в base64), чтобы поместить в раздел "исходники блоков". - Таблицы – можно либо представить как текст (например, вставив в текст страницы явным образом, возможно в Markdown-таблицу), либо хранить отдельно. Чтобы не потерять важные детали в Q&A, лучше **включить**

таблицы в текст в читабельном формате (например, каждую строку таблицы с новой строки, разделяя колонки символами, либо кратко описать таблицу). Дополнительно сам HTML таблицы сохранить как исходник блока.

На этом этапе получаем **полуструктурированные данные**: по каждой странице – последовательность блоков с их содержимым. Мы готовы перейти к высокоуровневому анализу содержимого.

Этап 3: Анализ содержимого (описание, вопросы, чанки) с помощью LLM

Теперь, обладая чистым текстом и описаниями, задействуем модель **LLM** для генерации требуемых текстовых итогов. Выбор модели критичен: нам нужен относительно мощный, но не слишком тяжёлый (<=12B) язык, *многоязычный* и обученный следовать инструкциям. Под эти критерии хорошо подходит, например, **Qwen-7B-Instruct** от Alibaba. Эта модель (~7 млрд параметров) обучена на нескольких языках (покрывает свыше 29 языков) и умеет точно следовать инструкциям, включая генерацию структурированных ответов (JSON) [9]. Также она оптимизирована под долгий контекст ~33k токенов [10], что полезно для обработки больших документов.

Основные задачи LLM в нашем конвейере:

1. Генерация описания документа. Мы передаём модели краткую сводку всего текста документа (или сам текст, если он помещается в контекст) с инструкцией: "Представь, что ты описываешь документ коллеге: что это за документ, о чём он?". Модель должна выдать связное **описание человеческим языком** – например: *"Это скан двухстраничного письма от банка, содержащее выписку по счёту. В документе перечислены транзакции за август 2023, есть логотип банка и подпись менеджера."* Такое описание дает общее понимание, не перечисляя всё подряд, а выделяя суть. Qwen-7B, будучи хорошо обученной на инструкции, сможет обобщить содержимое на русском или другом языке исходного документа. Если документ многоязычный, можно либо выбрать язык описания (например, по языку интерфейса – русский) либо сохранить язык оригинала – в зависимости от требований. В корпоративной среде часто нужно описание на языке пользователя системы.

2. Генерация вопросов-ответов (Q&A) по страницам. Затем LLM генерирует *контрольные вопросы* по каждой странице. Мы поочерёдно берем контент каждой страницы (текст + описания картинок на ней) и даём задачу модели: "Составь список вопросов, на которые можно ответить, изучив содержание страницы. Затем дай на каждый вопрос точный ответ по тексту." Модель, обладая способностями понимания текста, должна **самостоятельно придумать вопросы** и ответить на них, имитируя как бы проверку знаний по документу. Поскольку у нас может быть много информации на странице, просим сгенерировать достаточно вопросов (до 20-50). Вопросы должны покрывать все основные факты: от простых (например, "Кто подписал документ? – Иванов И.И.") до сложнее интегративных ("Какова общая сумма транзакций на странице? – 1 234 000 руб.").

Важно, что модель должна учитывать контекст всего документа. Например, если на стр.2 упоминается нечто, что раскрыто на стр.1, вопросы можно скорректировать, чтобы не получалось противоречий. Мы можем реализовать это, передавая модели сразу весь документ, но фокусируясь

на конкретной странице. Например: *"Вот полный документ. Теперь составь 20 вопросов и ответов по странице 2, учитывая информацию с других страниц, если нужно для контекста."* Таким образом LLM видит общую картину, но генерирует вопросы только по указанной странице. Это предотвратит дублирование одних и тех же вопросов для разных страниц и позволит задавать на стр.2 вопросы, которые требуют данные со стр.1 (если вдруг необходимо).

Модель Qwen умеет работать с подобными задачами, а её мультиязычность позволит генерировать вопросы на том же языке, что и документ (или на ином, как настроим). Например, для русскоязычного отчёта вопросы-ответы тоже будут на русском. Для документов на английском – на английском, и т.д.

(Замечание: генерация QA пар – распространённая методика для улучшения последующего поиска знаний. Современные LLM способны генерировать правдоподобные вопросы по тексту и отвечать на них, фактически выполняя роль экзаменатора по документу. Мы используем это, чтобы дать богатое представление данных документа в структурированном виде.)

Технически, вопросы-ответы можно попросить модель вернуть сразу в формате JSON (список объектов `{ "question": "...", "answer": "..." }`), чтобы упростить парсинг. Qwen-7B, судя по обзорам, как раз хорошо генерирует JSON-структуры по запросу [9]. Тем не менее, мы проверим и отвалидируем её выводы перед включением в финальный JSON.

3. Чанкирование текста для эмбеддингов. Параллельно или после QA мы готовим текстовые фрагменты (**чанки**) для системы поиска знаний (RAG). Скорее всего, это делается программно (не с помощью LLM, чтобы не тратить лишние токены). Мы можем воспользоваться модулем *unstructured.chunking* или написать свою функцию: пройти по объединённому тексту всего документа и разбить его на куски размером ~200-300 слов (или ~1000 символов) с небольшим overlapping (перекрытием) между соседними кусками для контекстной связанности. Разбиение стараемся делать по границам смысловых блоков – например, по абзацам или предложениям, чтобы не разорвать мысль. Для очень больших документов можно делать чанки по страницам (т.е. страница сама разбивается на 2-3 чанка, если она длинная). В результате получаем список строк (или абзацев) – **готовый набор для эмбеддинга**. Позже эти строки могут быть пропущены через модель эмбеддингов, занесены в векторный индекс и использоваться для ответов на запросы (это уже за рамками данного сервиса, но мы готовим почву).

Unstructured как библиотека ориентирована на подобные задачи RAG, поэтому её функциональность нам помогает: она обеспечивает очистку, нормализацию текста, а также может выдавать chunked элементы прямо со ссылками на оригинал [11]. Мы, в частности, сохраним соответствие каждого чанка с источником (например: чанки 5–7 соответствуют 2-й странице, абзацу такому-то). Это можно внести в JSON, если нужно (например, хранить в каждом чанке номер страницы в метаданных).

После выполнения LLM-шагов у нас готовы *три ключевых результата*: **описание документа**, **набор Q&A по каждой странице** и **список чанков текста**. Все они привязаны к исходному содержимому, которое тоже у нас сохранено по блокам. Переходим к формированию финальной структуры JSON.

Этап 4: Формирование структурированного JSON

На завершающем этапе собираем все полученные данные в единую JSON-структуру согласно требованиям. Возможная схема JSON:

```
{
  "document_id": "<идентификатор файла>",
  "document_description": "…",        // описание документа от LLM
  "full_text": "…",                   // полный текст документа с описаниями
изображений
  "pages": [                          // массив страниц с их подробностями
    {
      "page_number": 1,
      "text": "…",                    // полный текст страницы (подмножество
full_text)
      "questions_answers": [          // Q&A пары по стр.1
        {"question": "…", "answer": "…"},
        …
      ],
      "blocks": [                     // блоки страницы
        {
          "type": "text",
          "content": "…",             // текст блока
          "role": "paragraph_title/list/item/etc.",
          "coords": {"x":...], "y":...], "width":...], "height":...]}
        },
        {
          "type": "image",
          "content": "<base64 или ссылку>",
          "description": "…",        // сгенерированное описание изображения
          "coords": {...]}
        },
        {
          "type": "table",
          "content": "<HTML или CSV>",
          "coords": {...]}
        },
        …
      ],
      "structured_data": {...]}       // опционально: если Donut извлек поля
(напр., invoice)
    },
    { "page_number": 2, ... },
    …
  ],
  "chunks": [
```

```
    {"page": 1, "chunk_index": 1, "text": "…"},
    {"page": 1, "chunk_index": 2, "text": "…"},
    …
  ],
  "processing_metrics": {
    "upload_time_s": 1.234,
    "parsing_time_s": 2.345,
    "ocr_time_s": 3.210,
    "donut_time_s": 0.567,
    "llm_summary_time_s": 1.111,
    "llm_qa_time_s": 4.321,
    "total_time_s": 10.0,
    "ocr_engine": "PaddleOCR",
    "tokens_used_summary": 512,
    "tokens_used_qa": 2048,
    "cpu_usage_percent": 85.0,
    "gpu_memory_used_mb": 10240,
    ...
  }
}
```

(Приведённая схема приблизительна – в реализации поля могут немного отличаться названиями.)

Несколько моментов по содержанию JSON:

- **Описание и текст.** *Document_description* – строка с абстрактом. *Full_text* – полный текст всего документа. Хотя мы дублируем текст страниц внутри `pages[i].text`, включение целиком `full_text` удобно для некоторых случаев (например, быстро поискать по всему документу). Он формируется простым склеиванием текстов всех страниц в порядке. В него уже встроены описания изображений, т.е. если на стр.2 была печать и мы её описали как "[Печать: круглая, с гербом…]", это войдёт в `full_text` и в `pages[2].text`. Таким образом *full_text* – **обогащённый текст документа**, пригодный для чтения без визуального оригинала.

- **Q&A.** Структурированы по страницам, как массивы объектов. Мы включаем именно те вопросы, ответы на которые находятся *на этой странице*. Если какой-то вопрос охватывает несколько страниц, мы можем либо дублировать его в каждой из причастных страниц (что вряд ли нужно), либо отнести к наиболее релевантной странице. Предполагаем, что вопросы более локальны. Формат вопрос-ответ – строка-вопрос и строка-ответ. По желанию, можно добавить поле `source_text` с цитатой из текста, на основе которой дан ответ (для большей прозрачности), но этого не требовалось явным образом. Порядок вопросов – в порядке значимости или следования информации на странице.

- **Исходные блоки.** В массиве `blocks` храним все блоки как они были распознаны. *Type* позволяет отличить текст, изображение, таблицу и т.д. У текстовых блоков content – сам текст. У изображений content – либо base64-кодировка изображения, либо ссылка на сохранённый временный файл (например, `"file:///data/artifacts/img_<id>_p1_block3.png"`).

Дополнительно для изображений даём сгенерированное описание (`description`), которое по сути уже есть и в тексте страницы, но здесь оно явно связано с блоком. Можно сохранить и другие атрибуты – например, `confidence` OCR, шрифт, цвет – если это вдруг важно, но по заданию не требуется. Координаты (coords) блоков на странице храним, если нужно привязать обратно к оригиналу (например, для подсветки при отображении). Они могут быть в относительных единицах (проценты от ширины/высоты страницы) или в пикселях исходного изображения страницы.

- **Structured_data (Donut вывод)** – опциональный раздел в каждой странице, где храним JSON, полученный от Donut, если он применялся. Например, для страницы-счёта там будет объект с полями InvoiceId, VendorName и т.п. как показано в примере вывода Donut [7]. Если документ не подходил под известные шаблоны, этого раздела может не быть.

- **Chunks.** Массив чанков содержит каждый фрагмент текста и ссылается на страницу. Можно также добавить оффсет границы чанка внутри страницы или исходные блоки (например, id блоков, которые вошли в этот чанк). Это поможет, если нужно соотнести найденный через RAG кусок с оригиналом. Но минимально достаточно хранить текст чанка и номер страницы.

- **Метрики производительности.** Сюда мы собираем всевозможную статистику:

- Время выполнения основных стадий: загрузка, парсинг, OCR, инференс моделей (Donut, LLM summary, LLM QAs). Время измеряем с помощью таймеров (например, `time.time()` в Python вокруг вызовов) и округляем до мс.
- Количество использованных токенов каждой LLM: при вызове моделей через HuggingFace Transformers можно подсчитать длину prompt и длину сгенерированного ответа с помощью токенайзера. Например, у Qwen токенайзер нам даст число токенов в запросе описания и ответа. Эти цифры важны, чтобы контролировать объём задаваемого контекста и следить, не привысили ли лимит (в нашем случае Qwen2.5-7B позволяет до 33k контекста [10], что с запасом хватит, но учитывать всё равно полезно).
- Информация о загрузке CPU/GPU: можно взять среднюю загрузку CPU за время обработки (например, через `psutil`), объём занятой GPU памяти (через PyTorch, `torch.cuda.max_memory_allocated()`), и пр. Это скорее для внутренних целей мониторинга. В JSON мы их включаем по требованию – например, если надо отлаживать или собирать статистику по многим запускам.
- Название/версия используемых моделей: чтобы в случае обновления можно было проследить, какой моделью получен результат. Например, `llm_model: "Qwen-7B-instruct v2"`, `ocr_model: "PaddleOCR 2.6"` и т.д. (Это не было явно запрошено, но может оказаться полезным в отладке, так что можно добавить).

Сформировав JSON, сервис сохраняет его (например, во временный файл или в памяти) и помечает задание как завершённое. При обращении клиента к `/result/{id}` этот JSON отдается в ответ. После этого, если клиент посылает `/delete/{id}`, все файлы (исходник, изображения, JSON) удаляются.

Этап 5: Производительность, масштабируемость и очистка

Оптимизация ресурсов. Ограничение в 40 ГБ GPU-памяти диктует аккуратный выбор моделей и параллелизма. Мы выбрали модели до ~7–12B параметров; например, Qwen-7B занимает ~14–16 ГБ в памяти FP16. Мы можем применить *8-битное или 4-битное квантование* модели, чтобы сократить память без заметной потери качества [12]. 4-битная квантованная Qwen-7B может занимать ~5–6 ГБ, что позволяет одновременно загружать, скажем, Donut (~1–2 ГБ) и даже держать второй экземпляр модели для параллельной обработки другого документа. Donut-модель относительно компактна (она основана на Swin+BART [13], обычно сотни миллионов параметров, не миллиарды).

При последовательной обработке документов время исполнения ~линейно растёт с числом страниц. Чтобы масштабироваться, можно **запускать несколько контейнеров** этого сервиса (горизонтальное масштабирование) и распределять файлы между ними. Внутри одного контейнера также возможно параллелить обработку разных файлов потоками, но нужно избегать конкуренции за GPU: либо выполнять LLM-вызовы строго по одному за раз, либо разделить GPU на несколько логических устройств (например, с помощью NVIDIA Multi-Instance GPU, если доступно). Поскольку сервис без внешних зависимостей (весь inference локально), масштабирование достигается именно добавлением ресурсов (больше GPU машин).

Очистка данных. Мы предусматриваем автоматическую очистку файлов и промежуточных результатов. Каждый загруженный файл помечается меткой времени. Фоновый поток или при старте каждого нового запроса можно проверять директорию на наличие "просроченных" файлов (>3 дней) и удалять их. Аналогично удаляются связанные артефакты (распознанные изображения, кеши). Клиентский запрос на удаление форсирует немедленную очистку конкретного ID: сервис удалит сам файл, JSON, папку с временными изображениями и пр., и уберёт задание из очереди/памяти, если оно ещё там. Таким образом, данные не будут лежать дольше необходимого, что соответствует корпоративным требованиям безопасности.

Логирование и мониторинг. Сервис может вести журнал операций: когда файл получен, когда начата/закончена обработка, сколько времени заняло, и если были ошибки – стек трейс. Метрики из `processing_metrics` могут дублироваться в системный лог или в метрики Prometheus (при желании). Так ответственные лица смогут отслеживать среднее время обработки на страницу, процент загрузки GPU и заранее увеличивать мощности, если, например, среднее время растёт из-за перегрузки.

Отказоустойчивость. При сбое во время обработки (например, нехватка памяти, сбой модели) – сервис должен отловить исключение, сохранить статус "error" для данного файла и не упасть целиком. Клиент по `/status` увидит ошибку и может повторить запрос позже. В Docker-контейнер можно встроить монитор или просто рестарт политики orchestrator'a (docker-compose/k8s), чтобы при падении процесс перезапускался.

Обсуждение альтернатив и заключение

Мы рассмотрели вариант использования *мульти-агентного подхода*, когда один интеллект управляет разными инструментами. Например, концепция HuggingGPT предполагает, что LLM сам решает, какой моделью (OCR, VQA, т.д.) воспользоваться для разных частей задачи [14]. В теории это могло бы

упростить код – достаточно дать одному мощному агенту задание "вот документ, разберись и выдай JSON". Однако на практике такие подходы **пока недостаточно надёжны** для промышленного использования: сложно гарантировать, что агент выберет правильную стратегию и будет детерминированно работать каждый раз. К тому же, мощный мульти-модальный агент (типа GPT-4 Vision) – это облачный сервис, что не подходит для закрытой корпоративной среды, или огромная модель, не укладывающаяся в 40 ГБ.

Наш подход разбивает задачу на контролируемые шаги, каждый из которых выполняется оптимальным инструментом. Это упрощает отладку (мы на каждом этапе видим промежуточный результат – текст OCR, описания, и т.п.) и соответствие требованиям. Тем не менее, мы всё же воспользовались преимуществами мультимодальных моделей: например, Qwen-VL мы привлекаем для описания сложных изображений, а Qwen-7B для понимания текста – фактически, **работает связка моделей**, но она *явно* оркестрируется нашим кодом, а не скрыта внутри чёрного ящика.

Выбранные инструменты ориентированы на многоязычность и эффективность. **Qwen** семейство показало отличные результаты в китайском и английском, а также поддерживает множество других языков [9], что важно, ведь документы могут быть на русском, английском, и любом языке бизнеса. **Unstructured** обеспечивает единообразную обработку форматов документов и легко расширяется под новые типы файлов, плюс работает в контейнере, что подходит нашим условиям [1]. **Donut** избавляет от необходимости внедрять отдельный OCR движок для каждого языка и может повысить точность на сложных шаблонах документов [6]. Все эти компоненты с открытым исходным кодом, их можно развернуть локально без передачи данных вне корпоративной сети (важно для конфиденциальности).

Docker-контейнеризация обеспечивает переносимость: все зависимости (Python-библиотеки Unstructured, модели, OCR-data) упакованы вместе. Запуск на новом сервере сводится к `docker run -p 8000:8000 our_document_service`. При необходимости можно настроить GPU-доступ в контейнер (через NVIDIA Docker runtime) для ускорения.

В итоге, архитектура сочетает **классический ETL-подход** к документам (парсинг, извлечение структур) с **современными ML-моделями** для понимания содержания. Такое комбинированное решение (Unstructured + Donut + Qwen) позволяет получить из "сырого" документа максимально полное представление в JSON-формате. Все цели – от описания и полного текста до Q&A и статистики – достигаются в рамках заданных ограничений по ресурсам. За счёт модульности и логичного разбиения по этапам, система легко поддерживается и может развиваться: например, можно добавить новые модели (для рукописного ввода или для конкретных видов документов), обновить LLM на более мощную, либо масштабировать горизонтально под большую нагрузку. Это делает решение долгосрочно эффективным для использования внутри корпорации.

Источники:

- Unstructured – опенсорс-библиотека для конвертации документов в структурированные данные [1].
- Настройка OCR-агентов (Tesseract, PaddleOCR) в Unstructured для многоязычного распознавания [2].
- Donut – модель понимания документов без явного OCR, устойчиво работающая на разных языках и форматах [6].

- Применение Donut совместно с Unstructured для извлечения структурированных данных (кейсы счетов и квитанций) [8].
- Пример JSON, генерируемого моделью Donut для документа (счёт-фактура) [7].
- Модель Qwen-7B-Instruct – многоязычный LLM ~7 млрд параметров с улучшенным следованием инструкциям и генерацией JSON-структур [9].
- Мультимодальная модель Qwen-VL – обеспечивает передовой уровень понимания изображений (в том числе текст в них) на нескольких языках [15] [5].
- Методики ускорения инференса больших моделей с помощью квантования до int4 (снижение памяти) [12].
- Принцип работы агентных подходов (пример: HuggingGPT), где LLM разбивает задачу на подзадачи и вызывает специализированные модели [14].

---

[1] GitHub - Unstructured-IO/unstructured: Convert documents to structured data effortlessly. Unstructured is open-source ETL solution for transforming complex documents into clean, structured formats for language models. Visit our website to learn more about our enterprise grade Platform product for production grade workflows, partitioning, enrichments, chunking and embedding.
https://github.com/Unstructured-IO/unstructured

[2] Set the OCR agent - Unstructured
https://docs.unstructured.io/open-source/how-to/set-ocr-agent

[3] [11] Get chunked elements - Unstructured
https://docs.unstructured.io/open-source/how-to/get-chunked-elements

[4] [5] [15] Qwen/Qwen2-VL-7B-Instruct · Hugging Face
https://huggingface.co/Qwen/Qwen2-VL-7B-Instruct

[6] [12] [13] Donut
https://huggingface.co/docs/transformers/en/model_doc/donut

[7] [8] An Introduction to Vision Transformers for Document Understanding – Unstructured
https://unstructured.io/blog/an-introduction-to-vision-transformers-for-document-understanding

[9] [10] Qwen2.5-7B-Instruct | API, Fine-Tuning, Deployment
https://www.siliconflow.com/models/qwen-qwen2-5-7b-instruct

[14] [PDF] MMCTAgent: Multi-modal Critical Thinking Agent Framework for ...
https://www.microsoft.com/en-us/research/wp-content/uploads/2024/09/MMCT_NeurIPS_OWA_Final.pdf