28.05.2025 Планы

а, вот краткий и стратегический анализ текущих **планов по проекту Librarian AI** и **предлагаемые приоритеты** на ближайшие этапы:

**🔍 Обзор текущего состояния**

Вы реализовали:

* ✅ **Обработку файлов** (в том числе .zip/.rar/.tar)
* ✅ **Асинхронную очередь задач (Celery + Redis)** с контролем /status/{task\_id}
* ✅ **Потоковую загрузку больших файлов**
* ✅ **Генерацию мультиперспективных summary** с кастомными ролями и разными LLM (включая локальные модели)
* ✅ **LLMRouter** с поддержкой OpenRouter, GigaChat, Mistral, Dolphin
* ✅ **Архитектуру ядра с CLI, API и Telegram-адаптерами**

**🧭 Текущая фаза: Архитектурная стабилизация и расширение ИИ-ядра**

**📌 Краткие планы:**

| **Направление** | **Конкретные задачи** |
| --- | --- |
| ✅ **Универсальный LLMRouter** | ✔ GigaChat, ✔ Mistral, ✔ OpenRouter, ✔ Dolphin |
| 🧠 **Анализ и логические выводы** | Создание librarian\_ai.py, поддержка Chain-of-Thought |
| 📚 **База знаний** | Хранение чанков, графов, сущностей, feedback |
| 📈 **Панель мониторинга** | Статистика загрузок, LLM-ответов, кэширование |
| 🔐 **API-защита** | Внедрение API-ключей / JWT для безопасности |
| 🧪 **Тестирование** | Unit/Integration/Load тесты, мониторинг Celery/Redis |
| 🤖 **Игровые модели расследований** | Модели типа RedTeam vs BlueTeam, симуляции преступлений, выявление улик |
| ☁️ **Облачная стратегия** | Возможность запуска моделей в режиме "Low RAM", затем масштабирование с GPU |

**🧱 Архитектура моделей: приоритеты**

| **Категория** | **Цель** | **Приоритет** |
| --- | --- | --- |
| 🆓 Бесплатные LLM | GigaChat, OpenRouter, YandexGPT | ✅ Использовать сейчас |
| 🧠 Локальные LLM | Mistral 7B + дообучение на кейсах | 🚧 План на будущее с GPU |
| 🕵️ Табу-free модели | Dolphin, LM Studio | ✅ Уже подключено, расширить |
| 🧰 Fine-tuning | Обучение на спец. текстах (расследования) | 🚧 Позже, с финансированием |

**💼 Коммерческий потенциал**

| **Подход** | **Применение** | **Доход** |
| --- | --- | --- |
| 📦 SaaS API | Анализ документов, построение графов, детекция угроз | 💸 B2B (юристы, ИБ-компании) |
| 🧑‍💼 Виртуальные эксперты | Юрист, Психолог, Следователь | 💸 Платные консультации |
| 🎮 Образовательные симуляции | Расследование преступлений | 💡 Интерактивные курсы |
| 📖 Анализ книг, статей, архивов | Summary + знания | 💰 Подписка |

**🔧 Следующие шаги (по приоритету)**

1. **librarian\_ai.py** — логический Reasoner: выводы, гипотезы, действия (на основе графа и сущностей)
2. **Визуализация графа знаний** — через web/panel и CLI
3. **JWT-защита API** — разделение публичных и внутренних LLM-запросов
4. **Telegram-интерфейс** — быстрый анализ документов из телефона
5. **Интерактив CLI** — выбор модели, роли, режим анализа
6. **Кэширование LLM-ответов** (по prompt + hash)
7. **Расширение моделей (DeepSeek, Yi, LMStudio)**

**🧠 Заключение: перспектива**

Вы уже построили:

* Умную, модульную платформу
* С возможностью интеграции как SaaS, так и автономных приложений
* С гибкой архитектурой: локальные + облачные модели

👉 Ваш проект **готов к монетизации и масштабированию**, как только будет:

* Визуальный интерфейс (дешёвый MVP на Streamlit/FastAPI+Vue)
* Безопасность и стабильность (защита API, обработка ошибок)
* Первая версия Reasoner (логические выводы)

⚙️ Готов поддержать:

* Разработку librarian\_ai.py (Reasoning + выводы)
* Интеграцию новых LLM (DeepSeek, Yi, LMStudio)
* Подготовку MVP для инвесторов или клиентов

Для дальнейшего развития можно добавить:

* Поддержку паролей для архивов
* Распознавание текста в изображениях (OCR)
* Анализ вложений в email
* Интеграцию с облачными хранилищами (S3, GCS)

| **Методика** | **Ключевая идея** | **Плюсы** | **Минусы** | **Когда выбрать** |
| --- | --- | --- | --- | --- |
| **1. Монолит (расширенный OOP)** | Весь функционал (load, detect\_type, \_wrap\_stream, парсинг, OCR, валидация и т. д.) собирается в одном модуле или пакете. | • Простота разработки и деплоя (один пакет, одна точка входа) |  |  |
| • Низкие накладные расходы (нет сетевых вызовов) |  |  |  |  |
| • Лёгкий отладочный цикл в IDE | • Код быстро разрастается («God class») |  |  |  |
| • Трудно масштабировать отдельные участки |  |  |  |  |
| • Обновление всего — для мелких правок тоже нужно выкатывать монолит | Если объём проекта/трафика невелик и нет жёстких SLA, нужен быстрый MVP без сложной инфраструктуры. |  |  |  |
| **2. Плагинная архитектура** | Базовое ядро загружает плагины по интерфейсу LoaderPlugin, а специфическая логика формат–зависимой обработки вынесена в отдельные модули. | • Чистая инверсия зависимостей: ядро не знает о форматах |  |  |
| • Разные команды пишут плагины независимо |  |  |  |  |
| • Лёгкое включение/отключение новых форматов без правок ядра | • Нужно механизмы регистрации/поиска плагинов |  |  |  |
| • Тестировать множество комбинаций сложнее |  |  |  |  |
| • Всё ещё один процесс, нет ресурсной изоляции | Если ожидается частое добавление новых форматов или кастомная логика от разных команд, но нагрузка остаётся низкой. |  |  |  |
| **3. Микросервисы** | Каждый крупный функциональный блок (Upload, Detection, Extraction, OCR, Metadata) — свой HTTP/gRPC-сервис, за ним очередь/шина сообщений и API-шлюз. | • Горизонтальное масштабирование по компонентам |  |  |
| • Изоляция отказов: упали «PDF-сервис» — остальные работают |  |  |  |  |
| • Разные технологии в разных сервисах |  |  |  |  |
| • Плавное CI/CD и независимый релиз | • Сетевые задержки, сложность дебага распределёнки |  |  |  |
| • Необходимость оркестрации (Docker-Compose/K8s) |  |  |  |  |
| • Нужно централизованное логирование и мониторинг | Когда нагрузка растёт, нужна высокая отказоустойчивость, SLA по времени ответа, или множество команд работают над разными частями. |  |  |  |
| **4. Серверлесс / FaaS** | Локальная логика разбивается на «функции» (AWS Lambda, GCP Functions): детект, извлечение, OCR, каждая выкатывается отдельно и вызывается по API. | • Оплата «по факту», авто-масштабирование |  |  |
| • Не нужна собственная инфраструктура серверов |  |  |  |  |
| • Быстрое развёртывание «микро-функций» | • Ограничения времени работы (timeout), холодный старт |  |  |  |
| • Сложнее отлаживать локально |  |  |  |  |
| • Потребуется продуманная упаковка зависимостей | Если хотите минимизировать затраты на поддержание серверов и нагрузка «врязове-разовая», либо нужен гибкий burst-scale. |  |  |  |

**Рекомендации по выбору**

1. **Начало проекта или MVP**  
   – **Монолит** или **плагины**. Фокус на скорости разработки и выкате базовой функциональности.
2. **Рост функциональности**  
   – Если появляется много «точечных» требований от разных команд (новые форматы, облачные источники), переходите на **плагинную архитектуру** внутри единого процесса.
3. **Нарастающая нагрузка и SLA**  
   – Когда время ответа и отказоустойчивость критичны, переходите на **микросервисы**: разбейте SmartLoader на независимые сервисы, организуйте очередь задач и API-шлюз.
4. **Минимизация операционных расходов**  
   – Для нерегулярно используемых функций (OCR, тяжёлое ML-вычисление) имеет смысл переносить их в **серверлесс**.
5. **Гибридный подход**  
   – Часто применяется комбинация: ядро (detect, wrap\_stream) остаётся монолитом или плагином, тяжёлые операции (OCR, сложный парсинг) — отдельные микросервисы, а burst-задачи — в serverless.

**Итог**

* Если ваша цель — **быстро валидация идеи** и **единство кода**, выбирайте **монолит или плагины**.
* Если вы нацелены на **масштаб** и **надёжность**, сразу продумайте **микросервисы** с очередями и API-шлюзом.
* Для **всплеска потребления вычислительных ресурсов** можно гибко вынести тяжёлые задачи в **serverless**.

Таким образом, для дальнейшего развития вашего SmartLoader и системы «Либрари-ядра» наиболее естественным шагом будет поэтапный переход:

1. Монолит →
2. Плагины →
3. Микросервисы (+ опционально serverless для пиков).

Выберите ту ступень, которая соответствует вашим текущим требованиям по нагрузке и команде — и двигайтесь вместе с ростом проекта!

**Вы сказали:**

Начало проекта или MVP – Монолит или плагины. Фокус на скорости разработки и выкате базовой функциональности. Рост функциональности – Если появляется много «точечных» требований от разных команд (новые форматы, облачные источники), переходите на плагинную архитектуру внутри единого процесса. Нарастающая нагрузка и SLA – Когда время ответа и отказоустойчивость критичны, переходите на микросервисы: разбейте SmartLoader на независимые сервисы, организуйте очередь задач и API-шлюз. Минимизация операционных расходов – Для нерегулярно используемых функций (OCR, тяжёлое ML-вычисление) имеет смысл переносить их в серверлесс. Гибридный подход – Часто применяется комбинация: ядро (detect, wrap\_stream) остаётся монолитом или плагином, тяжёлые операции (OCR, сложный парсинг) — отдельные микросервисы, а burst-задачи — в serverless. посчитай время затраченное на переделывания вариантов системы

**ChatGPT сказал:**

Думал на протяжении 5 секунд

Ниже — примерная оценка трудозатрат на каждую из «стадий» переделки системы и суммарное время. В расчётах мы предполагаем одного опытного разработчика и включаем весь цикл: проектирование, реализация, тестирование и первичный deploy.

| **Стадия** | **Время** | **Комментарий** |
| --- | --- | --- |
| **1. Монолит (MVP)** | 2 недели | ~80 часов: базовый SmartLoader, детект, load, \_wrap\_stream и minimal CLI/API. |
| **2. Плагинная архитектура** | 1 неделя | ~40 часов: вынос логики форматов в плагины, регистрация/поиск плагинов, дока, тесты. |
| **3. Микросервисы** | 3 недели | ~120 часов: разбиение на сервисы (upload, detect, extract, metadata), очередь задач, gateway, docker-compose/k8s. |
| **4. Serverless-бэкенд для тяжёлых задач** | 1 неделя | ~40 часов: вынос OCR/ML-вызовов в функции Lambda/GCF, подготовка пакетов, CI, тесты «холодного старта». |
| **5. Гибридное объединение** | 1 неделя | ~40 часов: оркестрация микросервисов + serverless, интеграция кешей/логирования/мониторинга. |

**Сводка**

* **Монолит → плагины → микросервисы → serverless → гибрид**
* **Общая оценка:** ~320 часов (~8 недель полного рабочего времени)

Можно скорректировать:

* Если команда из 2–3 человек — чистое календарное время сократится (параллельная работа), но менеджмент коммуникаций добавит ~10–15 %.
* Если стартовый MVP «облегчённый» — монолит можно сделать за 1 неделю, суммарно выйдет ~240 ч (6 недель).

Эти цифры помогут спланировать ресурсы и понять, во сколько обойдётся полная рефакторинга системы по выбранным этапам.

недрение микросервисной архитектуры для вашего SmartLoader — отличная идея, особенно если вы планируете:

1. Обрабатывать файлы в распределенной системе
2. Масштабировать отдельные функции независимо
3. Использовать специализированные сервисы для разных типов файлов

**Архитектура микросервисного решения**

Copy

Download

┌─────────────────┐

│ API Gateway │

└─────────────────┘

│

┌───────────────┴───────────────┐

│ │

┌─────────────────────┐ ┌─────────────────────┐

│ File Upload MS │ │ File Analysis MS │

└─────────────────────┘ └─────────────────────┘

│ │

┌─────────────────────┐ ┌─────────────────────┐

│ Storage Service │ │ Metadata Service │

└─────────────────────┘ └─────────────────────┘

**1. Микросервис загрузки файлов (File Upload MS)**

**Реализация на FastAPI**:

python

Copy

Download

*# file\_upload\_ms/app.py*

from fastapi import FastAPI, UploadFile, HTTPException

from smart\_loader import SmartLoader

import aiofiles

app = FastAPI()

loader = SmartLoader()

@app.post("/upload")

async def upload\_file(file: UploadFile):

try:

*# Сохраняем во временный файл*

temp\_path = f"/tmp/{file.filename}"

async with aiofiles.open(temp\_path, 'wb') as out\_file:

content = await file.read()

await out\_file.write(content)

*# Обработка файла*

file\_type = loader.detect\_type(temp\_path)

return {"filename": file.filename, "type": file\_type}

except Exception as e:

raise HTTPException(status\_code=500, detail=str(e))

**2. Микросервис анализа файлов (File Analysis MS)**

**Реализация**:

python

Copy

Download

*# file\_analysis\_ms/app.py*

from fastapi import FastAPI

from pydantic import BaseModel

import httpx

app = FastAPI()

class AnalysisRequest(BaseModel):

file\_url: str

features: list[str] *# ['type', 'metadata', 'ocr']*

@app.post("/analyze")

async def analyze\_file(request: AnalysisRequest):

async with httpx.AsyncClient() as client:

*# Скачиваем файл*

resp = await client.get(request.file\_url)

resp.raise\_for\_status()

*# Анализ в зависимости от запрошенных функций*

results = {}

if 'metadata' in request.features:

results['metadata'] = await extract\_metadata(resp.content)

return results

**3. Сервис хранения (Storage Service)**

**Интеграция с S3-совместимым хранилищем**:

python

Copy

Download

*# storage\_service/app.py*

from fastapi import FastAPI, UploadFile

import boto3

from config import S3\_CONFIG

app = FastAPI()

s3 = boto3.client('s3', \*\*S3\_CONFIG)

@app.post("/store")

async def store\_file(file: UploadFile):

s3.upload\_fileobj(file.file, 'my-bucket', file.filename)

return {"url": f"s3://my-bucket/{file.filename}"}

**4. API Gateway**

**Конфигурация Traefik**:

yaml

Copy

Download

*# traefik.yml*

http:

routers:

upload:

rule: "PathPrefix(`/upload`)"

service: file-upload

analyze:

rule: "PathPrefix(`/analyze`)"

service: file-analysis

services:

file-upload:

loadBalancer:

servers:

- url: "http://file-upload-ms:8000"

file-analysis:

loadBalancer:

servers:

- url: "http://file-analysis-ms:8000"

**Развертывание с Docker**

**docker-compose.yml**:

yaml

Copy

Download

version: '3'

services:

api-gateway:

image: traefik:v2.5

ports:

- "80:80"

volumes:

- ./traefik.yml:/etc/traefik/traefik.yml

depends\_on:

- file-upload-ms

- file-analysis-ms

file-upload-ms:

build: ./file\_upload\_ms

ports:

- "8000:8000"

file-analysis-ms:

build: ./file\_analysis\_ms

ports:

- "8001:8000"

storage-service:

build: ./storage\_service

environment:

AWS\_ACCESS\_KEY\_ID: ${S3\_KEY}

AWS\_SECRET\_ACCESS\_KEY: ${S3\_SECRET}

ports:

- "8002:8000"

**Улучшения для микросервисной версии SmartLoader**

1. **Специализированные обработчики**:

python

Copy

Download

*# Выносим обработчики в отдельные сервисы*

FILE\_HANDLERS = {

'application/pdf': 'http://pdf-service:8000',

'image/': 'http://image-service:8000',

'text/': 'http://text-service:8000'

}

async def handle\_file(file\_input, mime\_type):

handler\_url = next((v for k, v in FILE\_HANDLERS.items()

if mime\_type.startswith(k)), None)

if handler\_url:

async with httpx.AsyncClient() as client:

files = {'file': file\_input}

return await client.post(f"{handler\_url}/process", files=files)

1. **Кэширование результатов**:

python

Copy

Download

*# Добавляем Redis для кэширования*

import redis

from fastapi\_cache import FastAPICache

from fastapi\_cache.backends.redis import RedisBackend

redis = redis.from\_url("redis://redis:6379")

FastAPICache.init(RedisBackend(redis), prefix="file-cache")

1. **Мониторинг и логирование**:

python

Copy

Download

*# Настройка OpenTelemetry*

from opentelemetry import trace

from opentelemetry.sdk.trace import TracerProvider

from opentelemetry.sdk.trace.export import BatchSpanProcessor

from opentelemetry.exporter.jaeger.thrift import JaegerExporter

trace.set\_tracer\_provider(TracerProvider())

tracer = trace.get\_tracer(\_\_name\_\_)

jaeger\_exporter = JaegerExporter(

agent\_host\_name="jaeger",

agent\_port=6831,

)

trace.get\_tracer\_provider().add\_span\_processor(

BatchSpanProcessor(jaeger\_exporter)

)

**Преимущества такого подхода**

1. **Горизонтальное масштабирование**:
   * Можно запускать несколько экземпляров каждого сервиса
   * Балансировка нагрузки через API Gateway
2. **Независимое развертывание**:
   * Обновлять анализатор PDF не влияя на обработку изображений
3. **Отказоустойчивость**:
   * Сервисы могут падать независимо
   * Возможность реализовать circuit breaker
4. **Гибкость технологий**:
   * Разные сервисы можно писать на разных языках
   * Например, сервис OCR на C++, а основной API на Python

**Рекомендации по внедрению**

1. Начните с выделения самого нагруженного функционала в отдельный сервис
2. Используйте асинхронные вызовы между сервисами
3. Реализуйте единый стандарт ошибок
4. Добавьте версионирование API с первого дня
5. Внедрите централизованное логирование и мониторинг

Какой аспект микросервисной архитектуры вас интересует больше всего? Хотите детально разобрать:

1. Взаимодействие между сервисами
2. Обработку ошибок в распределенной системе
3. Мониторинг и observability
4. Оптимизацию производительности
5. Безопасность микросервисов?

Ниже пример содержимого для каждого \_\_init\_\_.py в описанной вами структуре, чтобы пакеты корректно экспортировали нужные классы и модули.

text

КопироватьРедактировать

├── cli.py # новая «точка входа» вместо librarian\_ai.py

├── README.md

├── librarian\_ai/ # пакет с кодом

│ ├── \_\_init\_\_.py

│ ├── llm/

│ │ ├── \_\_init\_\_.py

│ │ └── llm\_router.py

│ └── tools/

│ ├── \_\_init\_\_.py

│ ├── embedder.py

│ └── loader.py

└── tests/

├── \_\_init\_\_.py

├── test\_embedder.py

└── test\_llm\_router\_pro.py

**1. librarian\_ai/\_\_init\_\_.py**

python

КопироватьРедактировать

# librarian\_ai/\_\_init\_\_.py

"""

Пакет librarian\_ai:

- Содержит код для работы с LLM (в папке llm)

- Утилиты (в папке tools)

- Экспорт «главных» классов и функций на верхнем уровне пакета

"""

\_\_version\_\_ = "0.1.0"

# === Экспорт классов/функций из llm ===

from .llm.llm\_router import (

LLMRouter,

LLMProvider,

default\_router,

query\_llm,

)

# Если у вас есть базовые клиенты в llm/base\_llm.py или local\_model.py, можно их тоже здесь экспортировать:

# from .llm.base\_llm import BaseLLMClient

# from .llm.local\_model import LocalModelClient

# === Экспорт утилит из tools ===

from .tools.embedder import Embedder, EmbeddingType, CacheInfo

from .tools.loader import SmartLoader

\_\_all\_\_ = [

# llm

"LLMRouter",

"LLMProvider",

"default\_router",

"query\_llm",

# "BaseLLMClient",

# "LocalModelClient",

# tools

"Embedder",

"EmbeddingType",

"CacheInfo",

"SmartLoader",

]

* Здесь мы явно экспортируем LLMRouter, LLMProvider, default\_router, query\_llm из модуля llm/llm\_router.py.
* Также экспортируются классы Embedder, EmbeddingType, CacheInfo из tools/embedder.py и SmartLoader из tools/loader.py.
* Если у вас найдутся другие «важные» классы (например, BaseLLMClient или LocalModelClient), их можно добавить в тот же \_\_all\_\_, после предварительного импорта.

**2. librarian\_ai/llm/\_\_init\_\_.py**

python

КопироватьРедактировать

# librarian\_ai/llm/\_\_init\_\_.py

"""

Подпакет llm:

Содержит маршрутизатор LLM и, при необходимости, другие классы-обёртки для разных провайдеров.

"""

# Экспортируем по умолчанию роутер, чтобы можно было писать:

# from librarian\_ai.llm import LLMRouter, LLMProvider

from .llm\_router import LLMRouter, LLMProvider, default\_router, query\_llm

\_\_all\_\_ = [

"LLMRouter",

"LLMProvider",

"default\_router",

"query\_llm",

]

* Благодаря этому, внутри тестов или кода достаточно писать:

python

КопироватьРедактировать

from librarian\_ai.llm import LLMRouter, LLMProvider

вместо длинного from librarian\_ai.llm.llm\_router import ....

**3. librarian\_ai/tools/\_\_init\_\_.py**

python

КопироватьРедактировать

# librarian\_ai/tools/\_\_init\_\_.py

"""

Подпакет tools:

Содержит все вспомогательные утилиты: векторизацию (embedder) и загрузку/анализ файлов (loader).

"""

from .embedder import Embedder, EmbeddingType, CacheInfo

from .loader import SmartLoader

\_\_all\_\_ = [

"Embedder",

"EmbeddingType",

"CacheInfo",

"SmartLoader",

]

* После этого достаточно писать:

python

КопироватьРедактировать

from librarian\_ai.tools import Embedder, SmartLoader

**4. tests/\_\_init\_\_.py**

python

КопироватьРедактировать

# tests/\_\_init\_\_.py

"""

Пакет для тестов.

Тестовые модули находятся в папке tests/ и имеют имена test\_\*.py или \*\_test.py.

"""

Обычно файл tests/\_\_init\_\_.py может быть пустым, но его наличие превращает tests/ в пакет, что упрощает импорт модулей вроде from librarian\_ai.tools import Embedder в тестах.

**5. Пример pytest.ini (в корне)**

ini

КопироватьРедактировать

# pytest.ini

[pytest]

# Папка, где pytest будет искать тесты

testpaths = tests

# Шаблоны имён файлов, которые считаются тестовыми

python\_files = test\_\*.py \*\_test.py

# Добавляем текущий каталог (корень проекта) в PYTHONPATH, чтобы пакет 'librarian\_ai' был виден

python\_paths = .

# Дополнительные опции pytest (короткий вывод, показывать причину пропусков)

addopts = -q -ra

* Благодаря python\_paths = . pytest сможет корректно обнаруживать пакет librarian\_ai и импортировать модули.

**6. Обновлённая структура проекта**

В итоге ваша папка Librarian AI должна выглядеть примерно так:

markdown

КопироватьРедактировать

G:\Librarian-AI\Librarian AI

├── cli.py # Новая точка входа (раньше это был librarian\_ai.py)

├── README.md

├── pytest.ini

├── requirements.txt

├── librarian\_ai/ # Пакет с кодом

│ ├── \_\_init\_\_.py

│ ├── llm/

│ │ ├── \_\_init\_\_.py

│ │ └── llm\_router.py

│ └── tools/

│ ├── \_\_init\_\_.py

│ ├── embedder.py

│ └── loader.py

└── tests/

├── \_\_init\_\_.py

├── test\_embedder.py

└── test\_llm\_router\_pro.py

* cli.py (или main.py) – это ваш новый файл-«точка входа» (вместо старого librarian\_ai.py).
* Пакет librarian\_ai/ содержит исходники, а tests/ – тесты.
* Благодаря настроенным \_\_init\_\_.py и pytest.ini импорты будут работать корректно.

После внесённых правок выполните в корне проекта:

bash

КопироватьРедактировать

# В активированном виртуальном окружении

pytest

И тесты должны собраться и выполниться без ошибок импорта.