**# 📁 Проект: Librarian AI (прототип агента-библиотекаря)**

**# ┌────────────────────────────────────────────┐**

**# |     Структура директорий и файлов v0.1    |**

**# └────────────────────────────────────────────┘**

librarian\_ai/

├── main.py                        # Главная точка запуска агента

├── install.py                     # 🚀 Установщик: создаёт структуру проекта

├── templates/                     # 🧩 Шаблоны начальных файлов

│   ├── main.py.tpl               # Шаблон main.py

│   ├── config.yaml.tpl           # Шаблон конфигурации

│   ├── base\_llm.py.tpl           # Шаблон базового LLM

│   ├── loader.py.tpl             # Шаблон загрузчика

│   └── readme.md.tpl             # Шаблон README

├── config/

│   ├── config.yaml               # Конфигурация агента (выбор ИИ, пути, параметры)

│   └── e\_full.yaml               # 📋 Полное описание агента в формате фермы (AgentDNA)

├── llm/

│   ├── base.py                   # Базовый интерфейс LLM

│   ├── yandex\_gpt.py             # Подключение к ЯндексGPT

│   ├── local\_chatglm.py          # Подключение к локальному ChatGLM

│   └── deep\_pavlov.py            # Подключение к DeepPavlov

├── core/

│   ├── loader.py                 # Загрузка документов и ссылок

│   ├── parser.py                 # Извлечение текста и метаданных

│   ├── classifier.py             # Классификация, теггинг, фильтрация

│   ├── embedder.py              # Векторизация текста (sentence-transformers, etc)

│   └── storage.py               # Работа с БД и базой знаний

├── db/

│   └── librarian.db              # SQLite база данных (или PostgreSQL)

├── knowledge/

│   └── vector\_store/             # Векторная база знаний (FAISS/Chroma)

├── utils/

│   └── logger.py                 # Логирование, диагностика, трассировка

├── cli/

│   └── agent\_cli.py              # CLI-интерфейс взаимодействия

├── telegram/

│   └── bot.py                    # Telegram-интерфейс (опционально)

├── agents/

│   ├── factory/                  # 📦 Структура фабрики агентов (будущие роли и шаблоны)

│   └── osint\_plus/               # 🛰️ OSINT+ агент (добыча информации из открытых источников)

│       ├── collector.py         # Основная логика сбора

│       ├── enrichers.py         # Модули обогащения данных (IP, соцсети, сайты)

│       ├── exporters.py         # Экспорт в JSON/базу

│       └── agent.yaml           # YAML-описание агента OSINT+

└── README.md                     # Документация проекта

**# Пример запуска:**

**# python main.py --mode batch --source ./data/news/**

**# python cli/agent\_cli.py --query "Что известно о компании X?"**

Файл main.py создан как отдельный документ и включает:

* 📄 Путь и имя
* 📌 Назначение
* 📥 Источники данных
* 📤 Что передаёт дальше

Продолжим с install.py?

# 📄 Файл: main.py | Расположение: librarian\_ai/main.py

# 📌 Назначение: Точка входа агента Librarian AI

# 📥 Получает: параметры из командной строки или по умолчанию

# 📤 Передаёт: команду в ядро для запуска агента, инициирует работу с БД и LLM

def main():

print("Librarian AI запущен.")

# TODO: Загрузить конфигурацию, инициализировать компоненты, запустить обработку данных

if \_\_name\_\_ == "\_\_main\_\_":

main()

Файл install.py создан с учётом твоего условия: все папки уже установлены вручную, и скрипт отвечает только за создание файлов на основе шаблонов.

Далее перейдём к config.yaml?

# 📄 Файл: install.py | Расположение: librarian\_ai/install.py

# 📌 Назначение: инициализация проекта Librarian AI — создание файлов из шаблонов

# 📥 Получает: шаблонные .tpl-файлы из папки templates/

# 📤 Передаёт: готовые .py, .yaml и .md файлы в нужные директории проекта

# ⚠️ Папки должны быть созданы заранее вручную

import os

file\_templates = {

"main.py": "templates/main.py.tpl",

"config/config.yaml": "templates/config.yaml.tpl",

"llm/base.py": "templates/base\_llm.py.tpl",

"core/loader.py": "templates/loader.py.tpl",

"README.md": "templates/readme.md.tpl",

"agents/osint\_plus/agent.yaml": "# YAML-описание OSINT агента\nname: osint\_plus\nversion: 0.1\nrole: collector\n",

"config/e\_full.yaml": "# Полное описание агента Librarian AI в формате AgentDNA\nname: librarian\_ai\nversion: 0.3\nrole: document\_librarian\ntags: [\"NLP\", \"RAG\", \"KnowledgeBase\"]\ndescription: |\n Агент для укладки, классификации и векторизации документов, сбора знаний и работы с ИИ.\nauthor: Viktor Kulichenko\norganization: AgentFarm Systems\ncompatibility: [\"OSINT+\", \"GigaChat\", \"YandexGPT\"]\n"

}

for path, tpl\_path in file\_templates.items():

full\_path = f"librarian\_ai/{path}"

if not os.path.exists(full\_path):

if tpl\_path.endswith(".tpl"):

with open(f"librarian\_ai/{tpl\_path}", "r", encoding="utf-8") as tpl\_file:

content = tpl\_file.read()

else:

content = tpl\_path or ""

with open(full\_path, "w", encoding="utf-8") as f:

f.write(content)

print("✅ Базовые файлы проекта Librarian AI созданы.")

Файл install.py создан с учётом твоего условия: все папки уже установлены вручную, и скрипт отвечает только за создание файлов на основе шаблонов.

Далее перейдём к config.yaml

# 📄 Файл: install.py | Расположение: librarian\_ai/install.py

# 📌 Назначение: инициализация проекта Librarian AI — создание файлов из шаблонов

# 📥 Получает: шаблонные .tpl-файлы из папки templates/

# 📤 Передаёт: готовые .py, .yaml и .md файлы в нужные директории проекта

# ⚠️ Папки должны быть созданы заранее вручную

import os

file\_templates = {

"main.py": "templates/main.py.tpl",

"config/config.yaml": "templates/config.yaml.tpl",

"llm/base.py": "templates/base\_llm.py.tpl",

"core/loader.py": "templates/loader.py.tpl",

"README.md": "templates/readme.md.tpl",

"agents/osint\_plus/agent.yaml": "# YAML-описание OSINT агента\nname: osint\_plus\nversion: 0.1\nrole: collector\n",

"config/e\_full.yaml": "# Полное описание агента Librarian AI в формате AgentDNA\nname: librarian\_ai\nversion: 0.3\nrole: document\_librarian\ntags: [\"NLP\", \"RAG\", \"KnowledgeBase\"]\ndescription: |\n Агент для укладки, классификации и векторизации документов, сбора знаний и работы с ИИ.\nauthor: Viktor Kulichenko\norganization: AgentFarm Systems\ncompatibility: [\"OSINT+\", \"GigaChat\", \"YandexGPT\"]\n"

}

for path, tpl\_path in file\_templates.items():

full\_path = f"librarian\_ai/{path}"

if not os.path.exists(full\_path):

if tpl\_path.endswith(".tpl"):

with open(f"librarian\_ai/{tpl\_path}", "r", encoding="utf-8") as tpl\_file:

content = tpl\_file.read()

else:

content = tpl\_path or ""

with open(full\_path, "w", encoding="utf-8") as f:

f.write(content)

print("✅ Базовые файлы проекта Librarian AI созданы.")

Файл config.yaml создан. Название агента зафиксировано как Librarian AI, переименование не предусмотрено, чтобы обеспечить стабильность в связке с другими агентами и ядром фермы.

Готов продолжать. Следующий — base.py в llm/?

# 📄 Файл: config.yaml | Расположение: librarian\_ai/config/config.yaml

# 📌 Назначение: конфигурация агента Librarian AI (название зафиксировано)

# 📥 Получает: настройки от разработчика или пользователя (LLM, пути, API)

# 📤 Передаёт: параметры в основной код и модули ядра

# Название агента — зафиксировано как Librarian AI, переименование не предусмотрено

name: librarian\_ai

version: 0.3

role: document\_librarian

# LLM-провайдер по умолчанию

llm\_provider: yandex\_gpt

# Пути к ресурсам

paths:

database: db/librarian.db

vector\_store: knowledge/vector\_store/

logs: logs/log.txt

# Включение дополнительных агентов

integrations:

osint\_plus: true

factory: true

# Язык по умолчанию

language: ru

Файл base.py создан. Он определяет абстрактный интерфейс BaseLLM для всех подключаемых ИИ. Все адаптеры (yandex\_gpt.py, chatglm.py и т.д.) будут его расширять.

Следующий файл — loader.py в core/?

Файл loader.py создан. Он отвечает за загрузку документов из указанной папки, фильтрует по расширениям и передаёт список файлов дальше в цепочку обработки.

Готов перейти к следующему — например, parser.py или yandex\_gpt.py?

📄 Файл: base.py | Расположение: librarian\_ai/llm/base.py

# 📌 Назначение: базовый интерфейс для всех LLM-провайдеров (Яндекс, ChatGLM, DeepPavlov)

# 📥 Получает: текст запроса от ядра агента

# 📤 Передаёт: сгенерированный ответ от LLM

from abc import ABC, abstractmethod

class BaseLLM(ABC):

"""

Абстрактный класс для подключения моделей LLM к Librarian AI.

Все адаптеры (yandex\_gpt.py, local\_chatglm.py и др.) должны наследовать этот интерфейс.

"""

@abstractmethod

def ask(self, prompt: str) -> str:

"""

Метод запроса к языковой модели.

:param prompt: входной запрос

:return: текстовый ответ от LLM

"""

pass

Файл parser.py создан. Он извлекает текст и базовую метаинформацию (имя, размер) из .txt-файлов. Позже можно подключить поддержку PDF, DOCX и HTML.

Переходим к yandex\_gpt.py?

# 📄 Файл: parser.py | Расположение: librarian\_ai/core/parser.py

# 📌 Назначение: извлечение текста и метаданных из файлов

# 📥 Получает: список путей к файлам (от loader.py)

# 📤 Передаёт: чистый текст и метаинформацию (название, дата, автор и пр.)

import os

# TODO: подключить парсеры PDF, DOCX, HTML при необходимости

def parse\_text\_file(filepath):

"""

Извлекает текст из обычного .txt файла.

:param filepath: путь к файлу

:return: словарь с текстом и метаинформацией

"""

try:

with open(filepath, "r", encoding="utf-8") as f:

text = f.read()

meta = {

"filename": os.path.basename(filepath),

"size\_kb": round(os.path.getsize(filepath) / 1024, 2)

}

return {"text": text, "meta": meta}

except Exception as e:

return {"error": str(e), "path": filepath}

if \_\_name\_\_ == "\_\_main\_\_":

from pprint import pprint

test = parse\_text\_file("../test\_data/example.txt")

pprint(test)

**🧠 Целевая структура RAG-платформы Librarian AI**

css

КопироватьРедактировать

Запрос пользователя

│

▼

[Query Router (Prompt + Intent)]

│

▼

┌────────────┐

│ Retriever │──► [Vector DB: QDrant / FAISS / Chroma]

└────────────┘

│

▼

[Re-Ranker] — улучшает релевантность

│

▼

[GraphRAG] — проверка связей и контекста

│

▼

[LLM Router (GPT / Yandex / DeepSeek)]

│

▼

Ответ пользователю

**✅ Что ты уже предусмотрел:**

* 🔗 Поддержку нескольких LLM через llm\_router.py
* 🔍 Векторную базу знаний (FAISS / Chroma)
* 📁 Структуру ingest → parse → classify → embed
* 🧩 Путь к GraphRAG (граф знаний + вектор)
* 🧠 Понимание проблемы Re-ranking и chunking

**🔧 Что можно добавить на следующем этапе:**

1. **retriever.py**  
   Компонент поиска релевантных фрагментов (на базе FAISS или QDrant)
2. **re\_ranker.py**  
   Использует CrossEncoder или модели rerank (например, bge-reranker, colbert)
3. **graph\_engine.py**  
   Поддержка GraphRAG — Neo4j, Arango или Nebula
4. **llamaindex\_connector.py**  
   Обёртка под LLamaIndex, если хочешь уйти от ручной сборки
5. **permissions.py**  
   Контроль доступа на уровне запроса и метаданных

📌 Хочешь — я выдам пример retriever.py и опишу, как внедрить Re-ranking и GraphRAG на твоей текущей архитектуре?

Или начнём с интеграции LLamaIndex + Graph + ReRank?

librarian\_ai/

├── main.py

├── install.py

├── templates/

│ ├── main.py.tpl

│ ├── config.yaml.tpl

│ └── ...

├── config/

│ ├── config.yaml # добавим llms\_enabled, default\_llm

│ └── e\_full.yaml

├── llm/

│ ├── base.py

│ ├── yandex\_gpt.py

│ ├── local\_chatglm.py

│ ├── deep\_pavlov.py

│ └── llm\_router.py # 🧠 выбирает нужную модель (GPT, DeepSeek, etc)

├── core/

│ ├── loader.py

│ ├── parser.py

│ ├── classifier.py

│ ├── embedder.py

│ ├── storage.py

│ ├── retriever.py # 🔍 поиск релевантных chunk’ов

│ └── re\_ranker.py # 🔁 переупорядочивание (Re-Ranking)

├── graph/

│ └── graph\_store.py # 📊 GraphRAG / Neo4j / Arango / Nebula

├── db/

│ └── librarian.db

├── knowledge/

│ ├── vector\_store/

│ └── graph\_cache/ # 🧠 Кэш графа знаний

├── utils/

│ └── logger.py

├── cli/

│ └── agent\_cli.py

├── telegram/

│ └── bot.py

├── agents/

│ ├── factory/

│ └── osint\_plus/

│ ├── collector.py

│ ├── enrichers.py

│ ├── exporters.py

│ └── agent.yaml

└── README.md

**📌 Комментарии к изменениям:**

* llm/llm\_router.py — централизованный мозг LLM
* core/retriever.py и core/re\_ranker.py — RAG-механизм
* graph/graph\_store.py — для GraphRAG (подключение к Neo4j, Arango, Nebula)
* knowledge/graph\_cache/ — сохранение связей в графовом формате

Готов приступить к созданию этих компонентов по порядку. Начать с llm\_router.py?

**Вы сказали:**

ДАЙ ПОЯСНЕНИЯ ДЛЯ КАЖДОГО ФАЙЛА

**ChatGPT сказал:**

Вот пояснения к каждому файлу и каталогу в новой структуре librarian\_ai/, включая всё, что необходимо для мультиLLM, RAG, Re-Ranking и GraphRAG-системы:

**📁 Корень проекта librarian\_ai/**

| **Файл / Папка** | **Назначение** |
| --- | --- |
| main.py | Точка входа в приложение. Запускает ядро, LLM, загрузку конфигурации |
| install.py | Скрипт, создающий шаблонные файлы (все папки создаются вручную) |
| README.md | Основная документация проекта |

**📁 templates/**

| **Файл** | **Назначение** |
| --- | --- |
| \*.tpl | Шаблоны кода, генерируемые install.py (например, main.py.tpl) |

**📁 config/**

| **Файл** | **Назначение** |
| --- | --- |
| config.yaml | Основной файл конфигурации агента (LLM, пути, интеграции, включения Graph, etc) |
| e\_full.yaml | Полное описание агента в формате AgentDNA (используется фабрикой агентов) |

**📁 llm/ — Модули языковых моделей**

| **Файл** | **Назначение** |
| --- | --- |
| base.py | Абстрактный интерфейс BaseLLM для подключения любых моделей |
| yandex\_gpt.py | Подключение к API ЯндексGPT |
| local\_chatglm.py | Работа с локальной моделью ChatGLM |
| deep\_pavlov.py | Интерфейс к DeepPavlov (возможно локально или через API) |
| llm\_router.py | Централизованный маршрутизатор между моделями. Выбор лучшей модели по типу запроса |

**📁 core/ — Ядро обработки**

| **Файл** | **Назначение** |
| --- | --- |
| loader.py | Загрузка файлов (PDF, TXT, DOCX) из указанной директории |
| parser.py | Извлечение текста и метаданных из файлов |
| classifier.py | Тематическая классификация, извлечение тегов, сущностей |
| embedder.py | Преобразование текста в векторные представления (sentence-transformers, etc) |
| storage.py | Хранение в реляционную БД |
| retriever.py | Компонент поиска релевантных chunk'ов из базы знаний (RAG) |
| re\_ranker.py | Re-ranking результатов семантического поиска, сортировка по релевантности |

**📁 graph/ — Граф знаний**

| **Файл** | **Назначение** |
| --- | --- |
| graph\_store.py | Работа с графовой базой (Neo4j, ArangoDB, Nebula). Обработка связей и сущностей |

**📁 knowledge/ — База знаний**

| **Папка / Файл** | **Назначение** |
| --- | --- |
| vector\_store/ | Векторное хранилище (FAISS, Chroma, QDrant и т.п.) |
| graph\_cache/ | Временный кэш графа знаний или индексируемых связей |

**📁 db/**

| **Файл** | **Назначение** |
| --- | --- |
| librarian.db | Локальная БД для хранения исходных текстов, метаданных, логов |

**📁 utils/**

| **Файл** | **Назначение** |
| --- | --- |
| logger.py | Модуль логирования: лог-файлы, отладка, трассировка действий |

**📁 cli/ — Командный интерфейс**

| **Файл** | **Назначение** |
| --- | --- |
| agent\_cli.py | Оболочка для запуска агентов, запроса к БЗ или LLM из консоли |

**📁 telegram/**

| **Файл** | **Назначение** |
| --- | --- |
| bot.py | Telegram-интерфейс агента. Принимает запросы, отправляет ответы |

**📁 agents/ — Дополнительные модули и цепочки агентов**

| **Папка** | **Назначение** |
| --- | --- |
| factory/ | Шаблоны агентов, создаваемых динамически (например, DataInsight, CISO и др.) |
| osint\_plus/ | Агент OSINT+ — сбор информации из открытых источников |
| collector.py | Сборщик URL, текстов, данных |
| enrichers.py | Распознавание IP, соцсетей, организаций |
| exporters.py | Сохранение результатов в JSON, CSV, БД |
| agent.yaml | YAML-описание OSINT-агента |

**🧱 # 📁 Проект: Librarian AI (прототип агента-библиотекаря)**

**# ┌────────────────────────────────────────────┐**

**# |     Структура директорий и файлов v0.1    |**

**# └────────────────────────────────────────────┘**

\*\*Librarian AI \*\*/

├── main.py                        # Главная точка запуска агента

├── install.py                     # 🚀 Установщик: создаёт структуру проекта

├── templates/                     # 🧹 Шаблоны начальных файлов

│       ├── main.py.tpl               # Шаблон main.py

│       ├── config.yaml.tpl           # Шаблон конфигурации

│       ├── base\_llm.py.tpl           # Шаблон базового LLM

│       ├── loader.py.tpl             # Шаблон загрузчика

│       └── readme.md.tpl             # Шаблон README

├── config/

│       ├── config.yaml               # Конфигурация агента (LLM, пути, включения)

│       └── e\_full.yaml               # 📋 Полное описание агента (AgentDNA)

├── llm/

│      ├── base.py                   # Базовой интерфейс LLM

│      ├─  yandex\_gpt.py             # Адаптер для YandexGPT

│      ├── local\_chatglm.py          # Адаптер для локальной ChatGLM

│      ├── deep\_pavlov.py            # Адаптер DeepPavlov

│      └── llm\_router.py             # 🧠 Маршрутизатор между моделями

├── core/

│      ├── loader.py                 # Загрузка документов

|      ├── parser.py                 # Извлечение текста и метаданных

│      ├── classifier.py             # Тематическая классификация

│      ├── embedder.py              # Векторизация (BERT, etc)

│      ├── storage.py               # Сохранение в базу данных

│      ├── retriever.py             # 🔍 Семантический поиск (RAG)

│      └── re\_ranker.py             # 🔁 Re-ranking документов по релевантности

├── graph/

│      └── graph\_store.py           # 📊 Обработка графа знаний (GraphRAG)

├── db/

│      └── librarian.db              # SQLite база данных

├── knowledge/

│        ├ ── vector\_store/            # Векторная база знаний

│        └── graph\_cache/             # Кэш графовой структуры

├── utils/

│          └── logger.py                # Логирование и диагностика

├── cli/

│           └── agent\_cli.py             # CLI-интерфейс для запросов

├── telegram/

│            └── bot.py                   # Telegram-бот (опционально)

├── agents/

│           ├── factory/                 # 🏠 Шаблоны и цепочки агентов

│           └── osint\_plus/              # 🛠️ OSINT+ агент

│                         ├── collector.py        # Сбор информации

│                         ├── enrichers.py        # Обогащение данных

│                         ├── exporters.py        # Экспорт в JSON/БД

│                         └── agent.yaml          # Паспорт описания OSINT агента

└── README.md                    # Основная документация

**# Пример запуска:**

**# python main.py --mode batch --source ./data/news/**

**# python cli/agent\_cli.py --query "Что известно о компании X?"**

**Следующие ключевые файлы (по порядку)**

**🔹 1. llm/llm\_router.py — мозг выбора LLM**

Позволит динамически переключаться между GPT, Yandex, ChatGLM и др.

**🔹 2. core/retriever.py — семантический поиск (RAG)**

Находит релевантные chunks по векторной базе (Chroma, FAISS, QDrant).

**🔹 3. core/re\_ranker.py — Re-ranking**

Улучшает порядок выдачи, подключает модели ранжирования.

**🔹 4. graph/graph\_store.py — граф знаний**

Позволяет использовать GraphRAG (связи между сущностями, Neo4j/Nebula).

**Дополнительно, если идём по слоям:**

**🛠 Поддержка векторной базы:**

* core/embedder.py — генерация векторов
* knowledge/vector\_store/ — как подключить FAISS / QDrant

**🔐 Безопасность и контроль:**

* utils/logger.py — логирование
* permissions.py (по желанию) — контроль доступа на уровне chunks

**🔄 Предлагаю порядок:**

1. llm\_router.py
2. retriever.py
3. re\_ranker.py
4. graph\_store.py
5. embedder.py

# 📄 Файл: llm\_router.py | Расположение: librarian\_ai/llm/llm\_router.py

# 📌 Назначение: маршрутизация между несколькими LLM (YandexGPT, ChatGLM, DeepPavlov)

# 📥 Получает: текст запроса от ядра агента

# 📤 Передаёт: ответ от выбранной модели

# Файл llm\_router.py готов ✅

# Он подключает и управляет несколькими LLM-модулями, включая YandexGPT, ChatGLM, DeepPavlov, с выбором по конфигурации.

from .yandex\_gpt import YandexGPT

from .local\_chatglm import ChatGLM

from .deep\_pavlov import DeepPavlov

class LLMRouter:

"""

Унифицированный интерфейс выбора модели для выполнения запроса.

Модель выбирается по умолчанию из конфигурации или явно передаётся при вызове.

"""

def \_\_init\_\_(self, config: dict):

self.models = {}

if config.get("llms\_enabled", {}).get("yandex", False):

self.models["yandex"] = YandexGPT(

api\_key=config.get("yandex\_api\_key", ""),

folder\_id=config.get("yandex\_folder\_id", "")

)

if config.get("llms\_enabled", {}).get("chatglm", False):

self.models["chatglm"] = ChatGLM()

if config.get("llms\_enabled", {}).get("pavlov", False):

self.models["pavlov"] = DeepPavlov()

self.default\_model = config.get("default\_llm", "yandex")

def ask(self, prompt: str, model: str = None) -> str:

"""

Выполняет запрос к выбранной модели

:param prompt: входной текст

:param model: (опц.) название модели ('yandex', 'chatglm', 'pavlov')

:return: текстовый ответ

"""

selected = model or self.default\_model

if selected not in self.models:

return f"[LLMRouter] ⚠️ Модель '{selected}' не подключена."

return self.models[selected].ask(prompt)

if \_\_name\_\_ == "\_\_main\_\_":

print("🔧 Тест LLMRouter (укажите конфигурацию вручную)...")

config = {

"llms\_enabled": {"yandex": True, "chatglm": False, "pavlov": False},

"yandex\_api\_key": "ваш\_API\_KEY",

"yandex\_folder\_id": "ваш\_FOLDER\_ID",

"default\_llm": "yandex"

}

router = LLMRouter(config)

print(router.ask("Привет, кто ты?"))

# 📄 Файл: test\_llm\_router\_pro.py | Расположение: tests/test\_llm\_router\_pro.py

# 📌 Назначение: модульное, интеграционное и нагрузочное тестирование LLMRouterPro

import unittest

from unittest.mock import MagicMock

from librarian\_ai.llm.llm\_router\_pro import LLMRouterPro

import threading

class TestLLMRouterPro(unittest.TestCase):

def setUp(self):

self.config = {

"llms\_enabled": {"yandex": True},

"yandex\_api\_key": "test",

"yandex\_folder\_id": "folder",

"default\_llm": "yandex",

"cache\_enabled": True

}

self.router = LLMRouterPro(self.config)

# Подменим настоящую модель на заглушку

mock\_model = MagicMock()

mock\_model.ask.return\_value = "ответ от заглушки"

self.router.models["yandex"] = mock\_model

def test\_ask\_default\_model(self):

result = self.router.ask("Привет")

self.assertEqual(result, "ответ от заглушки")

def test\_unknown\_model(self):

result = self.router.ask("Привет", model="несуществующая")

self.assertIn("не подключена", result)

def test\_async\_mode(self):

import asyncio

async def test():

response = await self.router.ask\_async("Что такое async?")

self.assertEqual(response, "ответ от заглушки")

asyncio.run(test())

def test\_metrics\_collected(self):

\_ = self.router.ask("Кто ты?")

self.assertIn("yandex", self.router.metrics)

self.assertTrue(len(self.router.metrics["yandex"]) > 0)

def test\_health\_check(self):

health = self.router.check\_health()

self.assertIn("yandex", health)

self.assertEqual(health["yandex"]["status"], "✅")

def test\_concurrent\_requests(self):

results = []

def worker():

result = self.router.ask("Тест нагрузки")

results.append(result)

threads = [threading.Thread(target=worker) for \_ in range(20)]

for t in threads:

t.start()

for t in threads:

t.join()

self.assertEqual(len(results), 20)

self.assertTrue(all(r == "ответ от заглушки" for r in results))

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**🚀 Следующие шаги**

**🔹 1. Создать модуль embedder.py**

Чтобы векторизовать тексты и построить FAISS-индекс. Это будет использоваться в retriever.py.

**Функции:**

* загрузка модели (sentence-transformers)
* преобразование текста в векторы
* сохранение индекса и метаданных

➡️ Можешь дать команду embedder.py, и я создам шаблон.

**🔹 2. Подготовить build\_index.py**

Скрипт, который:

* загружает документы из core/loader.py
* извлекает текст через parser.py
* генерирует эмбеддинги через embedder.py
* создаёт index.faiss и meta.pkl

➡️ Это первая фаза **pipeline подготовки знаний**.

**🔹 3. Тесты на Retriever + Embedder**

Создать модуль test\_retriever.py:

* Проверить, что retrieve() возвращает корректные chunks
* Замерить скорость и релевантность (возможно, мокать faiss)

**🔹 4. Связка с main.py**

В main.py прописать интеграционный режим:

bash

КопироватьРедактировать

python main.py --query "Как обеспечить ИБ в поликлинике?"

**🔹 5. Подключение Re-Ranker и GraphRAG (опц.)**

Добавим graph\_store.py и re\_ranker.py в ядро и обучим систему лучше понимать взаимосвязи в данных.

**🎯 Итого**

| **Этап** | **Цель** | **Статус** |
| --- | --- | --- |
| llm\_router\_pro.py | маршрутизатор | ✅ |
| retriever.py | поиск по базе | ✅ |
| embedder.py | векторизация | 🔜 |
| build\_index.py | генерация индекса | 🔜 |
| graph\_store.py | связи, GraphRAG | 🔜 |
| main.py | CLI-интерфейс | ⚙️ |
| test\_\* | надёжность | 🛠️ |

Конец формы

Да, модуль готов к запуску. Чтобы **выполнить тесты test\_llm\_router\_pro.py** локально, сделай следующее:

**✅ Запуск тестов (через Python)**

bash

КопироватьРедактировать

python -m unittest tests/test\_llm\_router\_pro.py

Если файл находится не в tests/, подставь актуальный путь.

**✅ Альтернатива: запуск через pytest (если установлен)**

bash

КопироватьРедактировать

pytest tests/test\_llm\_router\_pro.py -v

**🧪 Что проверяет:**

* ✅ обычный вызов модели
* ❌ вызов несуществующей модели
* 🔁 асинхронный режим
* 📊 сбор метрик
* ❤️ проверка работоспособности моделей
* 🚀 нагрузка (20 параллельных запросов)

Хочешь, чтобы я добавил автоматическую генерацию отчёта (например, html или junit.xml) после тестов?

**Вы сказали:**

какие щаги далее

**ChatGPT сказал:**

Отлично, у тебя уже есть:

✅ LLMRouterPro с продвинутой логикой  
✅ Тестовый модуль test\_llm\_router\_pro.py  
✅ Покрытие всех базовых и нагрузочных сценариев

**🚀 Следующие шаги**

**🔹 1. Создать модуль embedder.py**

Чтобы векторизовать тексты и построить FAISS-индекс. Это будет использоваться в retriever.py.

**Функции:**

* загрузка модели (sentence-transformers)
* преобразование текста в векторы
* сохранение индекса и метаданных

➡️ Можешь дать команду embedder.py, и я создам шаблон.

**🔹 2. Подготовить build\_index.py**

Скрипт, который:

* загружает документы из core/loader.py
* извлекает текст через parser.py
* генерирует эмбеддинги через embedder.py
* создаёт index.faiss и meta.pkl

➡️ Это первая фаза **pipeline подготовки знаний**.

**🔹 3. Тесты на Retriever + Embedder**

Создать модуль test\_retriever.py:

* Проверить, что retrieve() возвращает корректные chunks
* Замерить скорость и релевантность (возможно, мокать faiss)

**🔹 4. Связка с main.py**

В main.py прописать интеграционный режим:

bash

КопироватьРедактировать

python main.py --query "Как обеспечить ИБ в поликлинике?"

**🔹 5. Подключение Re-Ranker и GraphRAG (опц.)**

Добавим graph\_store.py и re\_ranker.py в ядро и обучим систему лучше понимать взаимосвязи в данных.

**🎯 Итого**

| **Этап** | **Цель** | **Статус** |
| --- | --- | --- |
| llm\_router\_pro.py | маршрутизатор | ✅ |
| retriever.py | поиск по базе | ✅ |
| embedder.py | векторизация | 🔜 |
| build\_index.py | генерация индекса | 🔜 |
| graph\_store.py | связи, GraphRAG | 🔜 |
| main.py | CLI-интерфейс | ⚙️ |
| test\_\* | надёжность | 🛠️ |

Хочешь — начну с embedder.py?

**Вы сказали:**

embedder.py

**ChatGPT сказал:**

Embedder

99

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

# 📄 Файл: embedder.py | Расположение: librarian\_ai/core/embedder.py

# 📌 Назначение: векторизация текстов и сохранение эмбеддингов для FAISS

# 📥 Получает: списки текстов (и метаданных)

# 📤 Передаёт: numpy-векторы, индекс FAISS и метаинформацию

from sentence\_transformers import SentenceTransformer

import numpy as np

import faiss

import os

import pickle

class Embedder:

def \_\_init\_\_(self, model\_name="all-MiniLM-L6-v2"):

self.model = SentenceTransformer(model\_name)

def encode(self, texts):

"""

Векторизация списка текстов

:param texts: List[str]

:return: np.array векторов

"""

return self.model.encode(texts, convert\_to\_numpy=True)

def save\_index(self, vectors, metadata, index\_path="knowledge/vector\_store/index.faiss", meta\_path="knowledge/vector\_store/meta.pkl"):

"""

Сохраняет FAISS-индекс и метаданные

:param vectors: np.array

:param metadata: List[dict]

"""

dim = vectors.shape[1]

index = faiss.IndexFlatL2(dim)

index.add(vectors)

faiss.write\_index(index, index\_path)

with open(meta\_path, "wb") as f:

pickle.dump(metadata, f)

Готово ✅  
Файл embedder.py создан. Он:

* использует sentence-transformers для векторизации,
* сохраняет FAISS-индекс и метаданные в knowledge/vector\_store/.

Следующий логичный шаг — build\_index.py, чтобы объединить loader, parser и embedder в единый pipeline генерации базы знаний. Продолжаем?

**Вы сказали:**

build\_index.py

**ChatGPT сказал:**

Build Index

99

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

# 📄 Файл: build\_index.py | Расположение: librarian\_ai/core/build\_index.py

# 📌 Назначение: генерация векторной базы из документов с сохранением индекса и метаданных

# 📥 Получает: путь к папке с файлами

# 📤 Передаёт: FAISS-индекс и метаданные для поиска через retriever

import os

from librarian\_ai.core.loader import load\_documents

from librarian\_ai.core.parser import parse\_text\_file

from librarian\_ai.core.embedder import Embedder

TEXT\_LIMIT = 3000 # максимум символов на chunk

def chunk\_text(text, size=TEXT\_LIMIT):

"""

Разбивает текст на куски размером size (для embedder)

"""

return [text[i:i+size] for i in range(0, len(text), size)]

def build\_index(source\_folder: str):

print(f"📂 Индексация данных из: {source\_folder}")

embedder = Embedder()

files = load\_documents(source\_folder)

all\_texts = []

metadata = []

for path in files:

parsed = parse\_text\_file(path)

if "text" not in parsed:

print(f"⚠️ Ошибка в {path}: {parsed.get('error')}")

continue

chunks = chunk\_text(parsed["text"])

all\_texts.extend(chunks)

Вместо этого ответить в чате

Готово ✅  
Скрипт build\_index.py создан. Он выполняет:

* загрузку и парсинг документов,
* разбиение текста на чанки,
* векторизацию и сохранение в FAISS + pickle.

Пример запуска:

bash

КопироватьРедактировать

python build\_index.py ./data/docs/

Вместо этого ответить в чате

Готово ✅  
Создан модуль graph\_store.py, который:

* подключается к Neo4j (или аналогичной базе),
* добавляет сущности и связи (GraphRAG),
* выполняет запросы по глубине,
* поддерживает очистку графа.

Если нужно — добавлю автоматический entity\_extractor.py, который будет извлекать имена, организации и ключевые темы из текста для наполнения графа. Продолжаем?

**👉 Основная функциональность:**

* **Создание узлов (add\_entity)** — добавляет новую сущность с заданным именем и типом узла (по умолчанию Concept).
* **Связывание сущностей (add\_relation)** — создает связь между двумя существующими узлами с указанием типа отношений.
* **Получение связанных сущностей (query\_related)** — выводит все смежные узлы для указанной сущности с возможностью ограничения глубины связей.
* **Очистка графа (clear\_graph)** — удаляет все существующие узлы и связи в базе данных.

**🧩 Архитектурные особенности:**

* **Использование Neo4j Driver**: Позволяет создавать соединения с базой данных, производить транзакционные операции и получать результаты запросов.
* **Типизированные ноды**: Каждый узел имеет имя и возможно дополнительные свойства, а также обозначается определенным лейблом (меткой).
* **Семантика отношений**: Каждая связь определяется типом отношения, что упрощает дальнейшие поиски и обработку.

**📊 Возможные улучшения и расширения функциональности:**

**1. Расширение типов сущностей**

Сейчас каждая новая сущность добавляется с одним и тем же лейблом («Concept»). Рекомендуется расширить набор возможных типов (лейблов), чтобы лучше структурировать хранилище и обеспечить больше возможностей для аналитики и обработки данных:

* Например, различать понятия и термины («Term»), алгоритмы («Algorithm»), инструменты («Tool») и др.

**2. Поддержка свойств сущностей**

Помимо имени, каждому узлу можно добавлять дополнительные характеристики (атрибуты). Это улучшит возможности дальнейшего анализа и обработки:

* Свойства могут включать авторство, популярность, использование в конкретных проектах и т.п.

**3. Контроль целостности данных**

Необходимо реализовать механизмы проверки существования и уникальности добавляемой сущности. Например, можно проверять, существует ли такой узел ранее, и либо обновить существующий, либо создать новый.

**4. Улучшенные запросы**

Расширьте метод query\_related, добавив возможность выводить не только имена, но и всю дополнительную информацию о связанной сущности (типа, атрибутах, путях и глубине связей). Также добавьте поддержку более продвинутых фильтров (по типу связи, атрибутам узлов и т.д.).

**5. Масштабируемость и производительность**

Если ожидается большое число операций над графом, подумайте о поддержке шардинга и репликации данных. Графовая база должна выдерживать большие нагрузки и обеспечивать высокую доступность.

**6. Визуализация и экспорт**

Реализуйте возможность вывода графической структуры данных (например, используя APOC Procedures для экспорта в графические форматы). Это упростит восприятие и понимание сложных взаимосвязей.

**🖋️ Практическое применение:**

Такой инструмент полезен в областях, где важно отслеживать взаимосвязи понятий и объектов:

* Информационная безопасность (модели угроз, атаки, методы защиты);
* Исследования и наука (создание семантического графа научных публикаций);
* Бизнес-аналитика (выявление зависимостей между отделами, проектами и ресурсами).

Пример использования:

store = GraphStore()

store.add\_entity("Машинное обучение", "Methodology")

store.add\_entity("Алгоритм Random Forest", "Algorithm")

store.add\_relation("Машинное обучение", "Алгоритм Random Forest", "ВКЛЮЧАЕТ")

print(store.query\_related("Машинное обучение"))

Это простое расширение предоставленной реализации значительно расширит спектр применений и повысит удобство использования данной системы.

**🔍 Что делает ReRanker?**

Основной задачей ReRanker является пересчёт релевантности уже отобранных кандидатом фрагментов текста с помощью специальной модели CrossEncoder. Этот подход улучшает качество выбора наиболее подходящих фрагментов для последующих этапов обработки, таких как генерация ответов искусственным интеллектом.

**Процесс работы:**

1. **Получаем кандидатные фрагменты** — Входящий список фрагментов документов формируется модулем поиска (retriever.py).
2. **Пересчитываем релевантность** — Используя модель CrossEncoder, оценивается вероятность того, насколько фрагмент соответствует запросу.
3. **Сортируем результаты** — Окончательно выбираются лучшие фрагменты по порядку убывания оценочной релевантности.

**🏷️ Ключевые элементы реализации:**

**Модель CrossEncoder**

Модель представляет собой предобученную нейронную сеть, предназначенную именно для точного сравнения пары "запрос-фрагмент". Она оценивает вероятность принадлежности конкретного фрагмента искомым результатам. Здесь используется версия MiniLM от Microsoft, которая демонстрирует хорошие показатели по соотношению производительности и точности.

**Код метода rerank**

Этот метод принимает три аргумента:

* **query**: сам запрос пользователя.
* **docs**: список фрагментов, полученных на предыдущем этапе поиска.
* **top\_k**: количество финально выбранных фрагментов.

Затем:

* Формируется список пар "запрос-текст" для передачи в модель.
* Рассчитываются рейтинги (scores) с помощью предсказания модели.
* Каждый документ обогащается полем "score", содержащим оценку релевантности.
* Документы сортируются по убыванию оценок, и возвращаются первые топ-K результатов.

# 📄 Файл: librarian\_ai

# 📂 Путь установки: librarian\_ai/core/re\_ranker.py

# 📌 Назначение: повторное ранжирование (Re-Ranking) результатов поиска

# 📥 Получает: список кандидатов от retriever (векторный поиск)

# ⚙️ Делает: пересчитывает релевантность с помощью CrossEncoder (или модели rerank)

# 📤 Передаёт: отсортированные по убыванию релевантности фрагменты (в LLM или визуализацию)

# 🚨 Потенциальные улучшения:

# 1. 📘 Выбор оптимальной модели CrossEncoder (например, ruBERT, DeepPavlov для РФ)

# 2. 🔄 Оптимизация: batched rerank, многопоточность, кэширование

# 3. 🇷🇺 Поддержка русского языка через мультиязычные модели

# 4. 🔬 Предобработка: удаление пунктуации, лемматизация, стоп-слова

from typing import List, Dict

from sentence\_transformers import CrossEncoder

import string

from pymorphy2 import MorphAnalyzer

morph = MorphAnalyzer()

class ReRanker:

def \_\_init\_\_(self, model\_name="DeepPavlov/rubert-base-cased-conversational"):

self.model = CrossEncoder(model\_name)

def lemmatize(self, word: str) -> str:

"""Приводит слово к нормальной форме."""

return morph.parse(word)[0].normal\_form

def preprocess(self, text: str) -> str:

"""Нормализация текста: лемматизация, очистка от пунктуации и цифр."""

words = text.split()

clean\_words = [self.lemmatize(w) for w in words if w.isalpha()]

return ' '.join(clean\_words)

def rerank(self, query: str, docs: List[Dict], top\_k: int = 5) -> List[Dict]:

"""

Ранжирует документы по релевантности.

:param query: исходный запрос

:param docs: список документов с ключом 'text'

:param top\_k: количество топ-документов

:return: список top\_k документов по убыванию релевантности

"""

query\_clean = self.preprocess(query)

pairs = [[query\_clean, self.preprocess(doc["text"])] for doc in docs]

scores = self.model.predict(pairs)

for i, score in enumerate(scores):

docs[i]["score"] = float(score)

return sorted(docs, key=lambda x: x["score"], reverse=True)[:top\_k]

if \_\_name\_\_ == "\_\_main\_\_":

reranker = ReRanker()

query = "обработка персональных данных"

candidates = [

{"text": "Персональные данные должны храниться согласно 152-ФЗ."},

{"text": "Шифрование — ключевой механизм защиты."},

{"text": "Персональные данные включают ФИО, СНИЛС и адрес проживания."}

]

top = reranker.rerank(query, candidates)

for doc in top:

print(f"⭐ {doc['score']:.4f} | {doc['text']}")

Конец формы

# 📄 Файл: entity\_extractor.py

# 📂 Путь установки: librarian\_ai/core/entity\_extractor.py

import re

import logging

import concurrent.futures

import natasha

import networkx as nx

import matplotlib.pyplot as plt

from collections import defaultdict

from dataclasses import dataclass

from typing import List, Dict, Optional, Tuple

try:

import spacy

except ImportError:

spacy = None

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

@dataclass

class Entity:

label: str

text: str

value: str

context: Optional[str] = None

confidence: Optional[float] = None

class EntityExtractor:

def \_\_init\_\_(self, lang: str = "ru", custom\_patterns: Optional[Dict[str, str]] = None,

enable\_multiprocessing: bool = True):

self.lang = lang

self.custom\_patterns = custom\_patterns or {}

self.enable\_multiprocessing = enable\_multiprocessing

self.\_init\_nlp\_models()

self.\_normalization\_cache = {}

self.extraction\_stats = defaultdict(int)

def \_init\_nlp\_models(self):

self.models = {}

try:

if self.lang == "ru":

self.segmenter = natasha.Segmenter()

self.morph\_vocab = natasha.MorphVocab()

self.models.update({

"names": natasha.NamesExtractor(self.morph\_vocab),

"orgs": natasha.OrgExtractor(self.morph\_vocab),

"addrs": natasha.AddrExtractor(self.morph\_vocab),

"dates": natasha.DateExtractor(self.morph\_vocab)

})

logger.info("Loaded Natasha models for Russian")

elif self.lang in ["en", "multi"]:

model\_name = "en\_core\_web\_lg" if self.lang == "en" else "xx\_ent\_wiki\_sm"

self.nlp = spacy.load(model\_name)

logger.info(f"Loaded spaCy model: {model\_name}")

else:

logger.warning(f"Unsupported language: {self.lang}")

except Exception as e:

logger.error(f"Failed to initialize NLP models: {e}")

self.models = {}

def \_extract\_with\_spacy(self, text: str) -> List[Entity]:

if not hasattr(self, 'nlp') or self.nlp is None:

return []

doc = self.nlp(text)

return [Entity(label=ent.label\_, text=ent.text, value=ent.text,

context=self.\_get\_context(text, ent.start\_char, ent.end\_char), confidence=0.9)

for ent in doc.ents]

def \_extract\_with\_natasha(self, text: str) -> List[Entity]:

if not self.models:

return []

doc = natasha.Doc(text)

doc.segment(self.segmenter)

entities = []

for name, extractor in self.models.items():

for match in extractor(text):

span = match.span

value = match.fact.as\_json if hasattr(match.fact, 'as\_json') else str(match.fact)

entities.append(Entity(

label=name.upper(),

text=text[span[0]:span[1]],

value=value,

context=self.\_get\_context(text, \*span),

confidence=0.85

))

return entities

def \_extract\_custom(self, text: str) -> List[Entity]:

entities = []

for label, pattern in self.custom\_patterns.items():

for match in re.finditer(pattern, text):

entities.append(Entity(

label=label,

text=match.group(),

value=match.group(),

context=self.\_get\_context(text, \*match.span()),

confidence=1.0

))

return entities

def \_get\_context(self, text: str, start: int, end: int, window: int = 50) -> str:

return text[max(0, start - window):min(len(text), end + window)]

def \_normalize\_entity(self, entity: Entity) -> Entity:

key = f"{entity.label}:{entity.text.lower()}"

if key in self.\_normalization\_cache:

return self.\_normalization\_cache[key]

if entity.label in ["ORG", "ORGS"]:

norm = re.sub(r'[^\w\s]', '', entity.value).strip()

norm = ' '.join(norm.split())

norm\_entity = Entity(label=entity.label, text=entity.text, value=norm,

context=entity.context, confidence=entity.confidence)

self.\_normalization\_cache[key] = norm\_entity

return norm\_entity

return entity

def \_post\_process(self, entities: List[Entity]) -> List[Entity]:

normalized = [self.\_normalize\_entity(e) for e in entities]

seen = set()

unique = []

for e in normalized:

key = (e.label, e.value.lower())

if key not in seen:

seen.add(key)

unique.append(e)

self.extraction\_stats[e.label] += 1

return unique

def extract\_entities(self, text: str) -> List[Entity]:

if not text or not isinstance(text, str):

logger.warning("Empty or invalid text provided")

return []

raw = []

if self.lang == "ru" and self.models:

raw += self.\_extract\_with\_natasha(text)

elif self.lang in ["en", "multi"] and hasattr(self, 'nlp'):

raw += self.\_extract\_with\_spacy(text)

raw += self.\_extract\_custom(text)

return self.\_post\_process(raw)

def batch\_extract(self, texts: List[str]) -> List[List[Entity]]:

if not self.enable\_multiprocessing or len(texts) < 5:

return [self.extract\_entities(t) for t in texts]

with concurrent.futures.ProcessPoolExecutor() as executor:

return list(executor.map(self.extract\_entities, texts))

def visualize\_entities(self, entities: List[Entity], filename: Optional[str] = None):

G = nx.Graph()

for i, e in enumerate(entities):

G.add\_node(i, label=e.label, text=e.text, value=e.value)

for i, e1 in enumerate(entities):

for j, e2 in enumerate(entities[i+1:], i+1):

if e1.context and e2.context and e1.context == e2.context:

G.add\_edge(i, j, relation="shared\_context")

pos = nx.spring\_layout(G)

plt.figure(figsize=(12, 12))

colors = [hash(G.nodes[n]['label']) % 20 for n in G.nodes()]

nx.draw(G, pos, node\_color=colors, with\_labels=True,

labels={n: f"{G.nodes[n]['label']}: {G.nodes[n]['text']}" for n in G.nodes()})

if filename:

plt.savefig(filename)

logger.info(f"Graph saved to {filename}")

else:

plt.show()

def get\_stats(self) -> Dict[str, int]:

return dict(self.extraction\_stats)

if \_\_name\_\_ == "\_\_main\_\_":

patterns = {

"DOCUMENT\_ID": r"\b\d{3}-[A-Z]{2}\b",

"PRODUCT\_CODE": r"\b[A-Z]{3}-\d{4}\b"

}

extractor = EntityExtractor(lang="en", custom\_patterns=patterns)

texts = [

"John Smith signed document 123-FE (Product ABC-1234) while working at Microsoft in New York.",

"Bill Gates founded Microsoft Corporation in 1975 (Document 456-GH).",

"The meeting discussed project XYZ-7890 with participants from Apple and IBM."

]

results = extractor.batch\_extract(texts)

for i, ents in enumerate(results):

print(f"\nText #{i+1}:")

for e in ents:

print(f"🔹 [{e.label}] {e.text} → {e.value} (Confidence: {e.confidence:.2f})")

extractor.visualize\_entities(results[0], "entities\_graph.png")

print("\nExtraction Statistics:")

for label, count in extractor.get\_stats().items():

print(f"{label}: {count}")

# 📄 Файл: entity\_extractor.py

# 📂 Путь установки: librarian\_ai/core/entity\_extractor.py

import re

import logging

import concurrent.futures

import natasha

import networkx as nx

import matplotlib.pyplot as plt

from collections import defaultdict

from dataclasses import dataclass

from typing import List, Dict, Optional, Tuple

try:

import spacy

except ImportError:

spacy = None

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

@dataclass

class Entity:

label: str

text: str

value: str

context: Optional[str] = None

confidence: Optional[float] = None

class EntityExtractor:

def \_\_init\_\_(self, lang: str = "ru", custom\_patterns: Optional[Dict[str, str]] = None,

enable\_multiprocessing: bool = True):

self.lang = lang

self.custom\_patterns = custom\_patterns or {}

self.enable\_multiprocessing = enable\_multiprocessing

self.\_init\_nlp\_models()

self.\_normalization\_cache = {}

self.extraction\_stats = defaultdict(int)

def \_init\_nlp\_models(self):

self.models = {}

try:

if self.lang == "ru":

self.segmenter = natasha.Segmenter()

self.morph\_vocab = natasha.MorphVocab()

self.models.update({

"names": natasha.NamesExtractor(self.morph\_vocab),

"orgs": natasha.OrgExtractor(self.morph\_vocab),

"addrs": natasha.AddrExtractor(self.morph\_vocab),

"dates": natasha.DateExtractor(self.morph\_vocab)

})

logger.info("Loaded Natasha models for Russian")

elif self.lang in ["en", "multi"]:

model\_name = "en\_core\_web\_lg" if self.lang == "en" else "xx\_ent\_wiki\_sm"

self.nlp = spacy.load(model\_name)

logger.info(f"Loaded spaCy model: {model\_name}")

else:

logger.warning(f"Unsupported language: {self.lang}")

except Exception as e:

logger.error(f"Failed to initialize NLP models: {e}")

self.models = {}

def \_extract\_with\_spacy(self, text: str) -> List[Entity]:

if not hasattr(self, 'nlp') or self.nlp is None:

return []

doc = self.nlp(text)

return [Entity(label=ent.label\_, text=ent.text, value=ent.text,

context=self.\_get\_context(text, ent.start\_char, ent.end\_char), confidence=0.9)

for ent in doc.ents]

def \_extract\_with\_natasha(self, text: str) -> List[Entity]:

if not self.models:

return []

doc = natasha.Doc(text)

doc.segment(self.segmenter)

entities = []

for name, extractor in self.models.items():

for match in extractor(text):

span = match.span

value = match.fact.as\_json if hasattr(match.fact, 'as\_json') else str(match.fact)

entities.append(Entity(

label=name.upper(),

text=text[span[0]:span[1]],

value=value,

context=self.\_get\_context(text, \*span),

confidence=0.85

))

return entities

def \_extract\_custom(self, text: str) -> List[Entity]:

entities = []

for label, pattern in self.custom\_patterns.items():

for match in re.finditer(pattern, text):

entities.append(Entity(

label=label,

text=match.group(),

value=match.group(),

context=self.\_get\_context(text, \*match.span()),

confidence=1.0

))

return entities

def \_get\_context(self, text: str, start: int, end: int, window: int = 50) -> str:

return text[max(0, start - window):min(len(text), end + window)]

def \_normalize\_entity(self, entity: Entity) -> Entity:

key = f"{entity.label}:{entity.text.lower()}"

if key in self.\_normalization\_cache:

return self.\_normalization\_cache[key]

if entity.label in ["ORG", "ORGS"]:

norm = re.sub(r'[^\w\s]', '', entity.value).strip()

norm = ' '.join(norm.split())

norm\_entity = Entity(label=entity.label, text=entity.text, value=norm,

context=entity.context, confidence=entity.confidence)

self.\_normalization\_cache[key] = norm\_entity

return norm\_entity

return entity

def \_post\_process(self, entities: List[Entity]) -> List[Entity]:

normalized = [self.\_normalize\_entity(e) for e in entities]

seen = set()

unique = []

for e in normalized:

key = (e.label, e.value.lower())

if key not in seen:

seen.add(key)

unique.append(e)

self.extraction\_stats[e.label] += 1

return unique

def extract\_entities(self, text: str) -> List[Entity]:

if not text or not isinstance(text, str):

logger.warning("Empty or invalid text provided")

return []

raw = []

if self.lang == "ru" and self.models:

raw += self.\_extract\_with\_natasha(text)

elif self.lang in ["en", "multi"] and hasattr(self, 'nlp'):

raw += self.\_extract\_with\_spacy(text)

raw += self.\_extract\_custom(text)

return self.\_post\_process(raw)

def batch\_extract(self, texts: List[str]) -> List[List[Entity]]:

if not self.enable\_multiprocessing or len(texts) < 5:

return [self.extract\_entities(t) for t in texts]

with concurrent.futures.ProcessPoolExecutor() as executor:

return list(executor.map(self.extract\_entities, texts))

def visualize\_entities(self, entities: List[Entity], filename: Optional[str] = None):

G = nx.Graph()

for i, e in enumerate(entities):

G.add\_node(i, label=e.label, text=e.text, value=e.value)

for i, e1 in enumerate(entities):

for j, e2 in enumerate(entities[i+1:], i+1):

if e1.context and e2.context and e1.context == e2.context:

G.add\_edge(i, j, relation="shared\_context")

pos = nx.spring\_layout(G)

plt.figure(figsize=(12, 12))

colors = [hash(G.nodes[n]['label']) % 20 for n in G.nodes()]

nx.draw(G, pos, node\_color=colors, with\_labels=True,

labels={n: f"{G.nodes[n]['label']}: {G.nodes[n]['text']}" for n in G.nodes()})

if filename:

plt.savefig(filename)

logger.info(f"Graph saved to {filename}")

else:

plt.show()

def get\_stats(self) -> Dict[str, int]:

return dict(self.extraction\_stats)

if \_\_name\_\_ == "\_\_main\_\_":

patterns = {

"DOCUMENT\_ID": r"\b\d{3}-[A-Z]{2}\b",

"PRODUCT\_CODE": r"\b[A-Z]{3}-\d{4}\b"

}

extractor = EntityExtractor(lang="en", custom\_patterns=patterns)

texts = [

"John Smith signed document 123-FE (Product ABC-1234) while working at Microsoft in New York.",

"Bill Gates founded Microsoft Corporation in 1975 (Document 456-GH).",

"The meeting discussed project XYZ-7890 with participants from Apple and IBM."

]

results = extractor.batch\_extract(texts)

for i, ents in enumerate(results):

print(f"\nText #{i+1}:")

for e in ents:

print(f"🔹 [{e.label}] {e.text} → {e.value} (Confidence: {e.confidence:.2f})")

extractor.visualize\_entities(results[0], "entities\_graph.png")

print("\nExtraction Statistics:")

for label, count in extractor.get\_stats().items():

print(f"{label}: {count}")

# 📄 Улучшенная версия entity\_extractor.py

# 📂 Путь: librarian\_ai/core/entity\_extractor.py

# ✅ Объединяет:

# - Расширенную инициализацию моделей (ru, en, multi, de, fr)

# - Поддержку пользовательских паттернов

# - Нормализацию сущностей

# - Постобработку и статистику

# - Визуализацию графа знаний

# - Пример запуска в \_\_main\_\_

# ⏭ Готова к интеграции ML-моделей, async и БД

# 📌 Используется в Librarian AI

# 📥 Получает: список текстов / один текст

# 📤 Возвращает: список нормализованных сущностей

# 🧠 NLP: Natasha / spaCy / Regex

# 🧪 Тесты: встроены в блок if \_\_name\_\_ == "\_\_main\_\_"

# 📄 Файл: entity\_extractor.py

# 📂 Путь установки: librarian\_ai/core/entity\_extractor.py

import re

import logging

import concurrent.futures

import natasha

import networkx as nx

import matplotlib.pyplot as plt

from collections import defaultdict

from dataclasses import dataclass

from typing import List, Dict, Optional, Tuple

try:

    import spacy

except ImportError:

    spacy = None

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

@dataclass

class Entity:

    label: str

    text: str

    value: str

    context: Optional[str] = None

    confidence: Optional[float] = None

class EntityExtractor:

    def \_\_init\_\_(self, lang: str = "ru", custom\_patterns: Optional[Dict[str, str]] = None,

                 enable\_multiprocessing: bool = True):

        self.lang = lang

        self.custom\_patterns = custom\_patterns or {}

        self.enable\_multiprocessing = enable\_multiprocessing

        self.\_init\_nlp\_models()

        self.\_normalization\_cache = {}

        self.extraction\_stats = defaultdict(int)

    def \_init\_nlp\_models(self):

        self.models = {}

        try:

            if self.lang == "ru":

                try:

                    from natasha import Segmenter, MorphVocab, NamesExtractor, AddrExtractor, OrgExtractor, DateExtractor

                    self.segmenter = Segmenter()

                    self.morph\_vocab = MorphVocab()

                    self.models = {

                        "PERSON": NamesExtractor(self.morph\_vocab),

                        "ORG": OrgExtractor(self.morph\_vocab),

                        "LOC": AddrExtractor(self.morph\_vocab),

                        "DATE": DateExtractor(self.morph\_vocab)

                    }

                    logger.info("Natasha models loaded successfully")

                except ImportError:

                    logger.warning("Natasha not available, falling back to regex only")

                    self.models = {}

            elif self.lang in ["en", "multi"]:

                model\_name = "en\_core\_web\_sm" if self.lang == "en" else "xx\_ent\_wiki\_sm"

                try:

                    import spacy

                    try:

                        self.nlp = spacy.load(model\_name)

                    except OSError:

                        logger.warning(f"spaCy model {model\_name} not found, downloading...")

                        from spacy.cli import download

                        download(model\_name)

                        self.nlp = spacy.load(model\_name)

                except ImportError:

                    logger.warning("spaCy not available, fallback to regex only")

                    self.nlp = None

            elif self.lang == "de":

                try:

                    import spacy

                    self.nlp = spacy.load("de\_core\_news\_sm")

                    logger.info("Loaded German spaCy model")

                except:

                    logger.warning("German model not available")

            elif self.lang == "fr":

                try:

                    import spacy

                    self.nlp = spacy.load("fr\_core\_news\_sm")

                    logger.info("Loaded French spaCy model")

                except:

                    logger.warning("French model not available")

        except Exception as e:

            logger.error(f"Critical error during model initialization: {str(e)}")

            self.models = {}

            self.nlp = None

    def \_extract\_with\_spacy(self, text: str) -> List[Entity]:

        if not hasattr(self, 'nlp') or self.nlp is None:

            return []

        doc = self.nlp(text)

        return [Entity(label=ent.label\_, text=ent.text, value=ent.text,

                       context=self.\_get\_context(text, ent.start\_char, ent.end\_char), confidence=0.9)

                for ent in doc.ents]

    def \_extract\_with\_natasha(self, text: str) -> List[Entity]:

        if not self.models:

            return []

        doc = natasha.Doc(text)

        doc.segment(self.segmenter)

        entities = []

        for label, extractor in self.models.items():

            for match in extractor(text):

                span = match.span

                value = match.fact.as\_json if hasattr(match.fact, 'as\_json') else str(match.fact)

                entities.append(Entity(

                    label=label,

                    text=text[span[0]:span[1]],

                    value=value,

                    context=self.\_get\_context(text, \*span),

                    confidence=0.85

                ))

        return entities

    def \_extract\_custom(self, text: str) -> List[Entity]:

        entities = []

        for label, pattern in self.custom\_patterns.items():

            for match in re.finditer(pattern, text):

                entities.append(Entity(

                    label=label,

                    text=match.group(),

                    value=match.group(),

                    context=self.\_get\_context(text, \*match.span()),

                    confidence=1.0

                ))

        return entities

    def \_get\_context(self, text: str, start: int, end: int, window: int = 50) -> str:

        return text[max(0, start - window):min(len(text), end + window)]

    def \_normalize\_entity(self, entity: Entity) -> Entity:

        cache\_key = f"{entity.label}:{entity.text.lower()}"

        if cache\_key in self.\_normalization\_cache:

            return self.\_normalization\_cache[cache\_key]

        norm\_text = entity.text

        norm\_value = entity.value

        if entity.label in ["PERSON", "PER"]:

            parts = [p.strip() for p in re.split(r'\s+', norm\_value) if p.strip()]

            if len(parts) > 1:

                norm\_value = f"{parts[0]} {'.'.join([p[0] for p in parts[1:] if p])}."

        elif entity.label in ["ORG", "ORGANIZATION"]:

            norm\_value = re.sub(r'\b(LLC|Inc|Corp|Ltd|ООО|АО|ЗАО)\b', '', norm\_value, flags=re.IGNORECASE)

            norm\_value = re.sub(r'[^\w\s]', '', norm\_value).strip().title()

        elif entity.label == "DATE":

            try:

                from dateutil.parser import parse

                dt = parse(norm\_value)

                norm\_value = dt.strftime("%Y-%m-%d")

            except:

                pass

        norm\_entity = Entity(

            label=entity.label,

            text=norm\_text,

            value=norm\_value,

            context=entity.context,

            confidence=entity.confidence

        )

        self.\_normalization\_cache[cache\_key] = norm\_entity

        return norm\_entity

    def \_post\_process(self, entities: List[Entity]) -> List[Entity]:

        normalized = [self.\_normalize\_entity(e) for e in entities]

        seen = set()

        unique = []

        for e in normalized:

            key = (e.label, e.value.lower())

            if key not in seen:

                seen.add(key)

                unique.append(e)

                self.extraction\_stats[e.label] += 1

        return unique

    def extract\_entities(self, text: str) -> List[Entity]:

        import time

        logger.info(f"Starting entity extraction for text: {text[:50]}...")

        start\_time = time.time()

        try:

            raw = []

            if self.lang == "ru" and self.models:

                raw += self.\_extract\_with\_natasha(text)

            elif self.lang in ["en", "multi", "de", "fr"] and hasattr(self, 'nlp'):

                raw += self.\_extract\_with\_spacy(text)

            raw += self.\_extract\_custom(text)

            processed = self.\_post\_process(raw)

            logger.info(f"Extracted {len(processed)} entities in {time.time() - start\_time:.2f}s")

            return processed

        except Exception as e:

            logger.error(f"Error during entity extraction: {str(e)}")

            return []

    def batch\_extract(self, texts: List[str]) -> List[List[Entity]]:

        if not self.enable\_multiprocessing or len(texts) < 5:

            return [self.extract\_entities(t) for t in texts]

        with concurrent.futures.ProcessPoolExecutor() as executor:

            return list(executor.map(self.extract\_entities, texts))

    def visualize\_entities(self, entities: List[Entity], filename: Optional[str] = None):

        G = nx.Graph()

        for i, e in enumerate(entities):

            G.add\_node(i, label=e.label, text=e.text, value=e.value)

        for i, e1 in enumerate(entities):

            for j, e2 in enumerate(entities[i+1:], i+1):

                if e1.context and e2.context and e1.context == e2.context:

                    G.add\_edge(i, j, relation="shared\_context")

        pos = nx.spring\_layout(G)

        plt.figure(figsize=(12, 12))

        colors = [hash(G.nodes[n]['label']) % 20 for n in G.nodes()]

        nx.draw(G, pos, node\_color=colors, with\_labels=True,

                labels={n: f"{G.nodes[n]['label']}: {G.nodes[n]['text']}" for n in G.nodes()})

        if filename:

            plt.savefig(filename)

            logger.info(f"Graph saved to {filename}")

        else:

            plt.show()

    def get\_stats(self) -> Dict[str, int]:

        return dict(self.extraction\_stats)

if \_\_name\_\_ == "\_\_main\_\_":

    patterns = {

        "DOCUMENT\_ID": r"\b\d{3}-[A-Z]{2}\b",

        "PRODUCT\_CODE": r"\b[A-Z]{3}-\d{4}\b"

    }

    extractor = EntityExtractor(lang="en", custom\_patterns=patterns)

    texts = [

        "John Smith signed document 123-FE (Product ABC-1234) while working at Microsoft in New York.",

        "Bill Gates founded Microsoft Corporation in 1975 (Document 456-GH).",

        "The meeting discussed project XYZ-7890 with participants from Apple and IBM."

    ]

    results = extractor.batch\_extract(texts)

    for i, ents in enumerate(results):

        print(f"\nText #{i+1}:")

        for e in ents:

            print(f"🔹 [{e.label}] {e.text} → {e.value} (Confidence: {e.confidence:.2f})")

    extractor.visualize\_entities(results[0], "entities\_graph.png")

    print("\nExtraction Statistics:")

    for label, count in extractor.get\_stats().items():

        print(f"{label}: {count}")

Отлично, entity\_extractor.py установлен. Далее мы последовательно создадим и установим все связанные с ним модули. Они группируются по следующим логическим зонам:

**📁 Модули, связанные с entity\_extractor.py**

**1. 🔧 Инфраструктура сущностей**

* entity.py — структура dataclass Entity (если будет вынесен отдельно).
* normalizer.py *(опционально)* — нормализаторы для сущностей.
* patterns.yaml *(опционально)* — словарь регулярных выражений для пользовательских сущностей.

**2. 📦 Визуализация и графы**

* visualizer.py — модуль визуализации графов сущностей на основе NetworkX и Matplotlib.

**3. 🧪 Тестирование**

* test\_entity\_extractor.py — модульные тесты на разные языки, шаблоны и классы сущностей.

**4. ⚙️ Конфигурация и запуск**

* config.json или config.yaml — конфигурация EntityExtractor.
* run\_extraction.py — простой CLI-скрипт для запуска на файле или директории.

**С чего начнём?**

Выберите, какой модуль создать следующим:

1. visualizer.py — визуализация графа отдельно.
2. test\_entity\_extractor.py — модульные тесты.
3. run\_extraction.py — скрипт запуска.
4. patterns.yaml — словарь паттернов.
5. entity.py — вынос Entity в отдельный dataclass.
6. Всё по очереди — с visualizer.py.

Что предпочтительнее?

# 📄 Файл: visualizer.py

# 📂 Путь установки: librarian\_ai/core/visualizer.py

# 📌 Назначение: визуализация графа сущностей и их контекстных связей

import networkx as nx

import matplotlib.pyplot as plt

from typing import List, Optional

from entity\_extractor import Entity # предполагаем, что Entity импортируем из основного модуля

def visualize\_entities(entities: List[Entity], filename: Optional[str] = None, title: str = "Entity Graph"):

"""

Визуализирует граф сущностей с контекстной связью.

:param entities: список сущностей

:param filename: если указано — сохраняет граф в файл, иначе отображает

:param title: заголовок графа

"""

G = nx.Graph()

# Добавляем узлы

for i, entity in enumerate(entities):

G.add\_node(i, label=entity.label, text=entity.text, value=entity.value)

# Добавляем связи (если у сущностей одинаковый контекст — считаем связанными)

for i, e1 in enumerate(entities):

for j, e2 in enumerate(entities[i + 1:], i + 1):

if e1.context and e2.context and e1.context == e2.context:

G.add\_edge(i, j, relation="shared\_context")

# Визуализация

pos = nx.spring\_layout(G, seed=42)

plt.figure(figsize=(12, 12))

colors = [hash(G.nodes[n]['label']) % 20 for n in G.nodes()]

nx.draw(

G, pos,

node\_color=colors,

with\_labels=True,

labels={n: f"{G.nodes[n]['label']}:\n{G.nodes[n]['text']}" for n in G.nodes()},

font\_size=9,

node\_size=2000

)

plt.title(title)

if filename:

plt.savefig(filename)

print(f"✅ Граф сохранён в файл: {filename}")

else:

plt.show()

✅ Возможности:

Раскраска по типу сущностей (через hash(label) % 20)

Указание filename сохраняет граф в .png

Использует context как связь между сущностями

**🔭 1. Интеграция с визуализатором visualizer.py**

**Цель:** автоматическая визуализация результатов после извлечения сущностей.

**Действия:**

* Добавить метод to\_graph() в EntityExtractor, возвращающий nx.Graph.
* Подключить EntityVisualizer из visualizer.py для визуализации графа сущностей.
* Создать CLI или Web-интерфейс, где пользователь видит граф после анализа текста.

**🔌 2. Поддержка RAG-системы и нейросетевого диалога**

**Цель:** использовать извлечённые сущности как элементы запроса в Retrieval-Augmented Generation.

**Действия:**

* Создать модуль rag\_router.py, который будет:
  + принимать текст → выделять сущности → запрашивать по ним документы.
* Встраивать извлечённые сущности в prompt к LLM (например: "Что вы знаете о {ORG} в контексте {DATE}?").
* Интегрировать с LangChain, LlamaIndex или Semantic Kernel.

**🔄 3. Автоматическая обработка больших документов**

**Цель:** обрабатывать пачки PDF, DOCX, TXT → извлекать сущности → формировать отчёты и графы.

**Действия:**

* Модуль document\_loader.py для разбора документов.
* Разметка сущностей + визуализация + экспорт отчёта (CSV, JSON, GEXF).
* Добавить логирование на каждый файл.

**⚙️ 4. Интерфейс (CLI / Web / API)**

**Цель:** удобный способ взаимодействия с системой без прямого редактирования кода.

**Варианты:**

* CLI: python run\_extractor.py --input file.txt --config config.yaml
* FastAPI: REST API для загрузки текста, получения сущностей и графа.
* Streamlit или Gradio: визуальный интерфейс для текста и графа.

**🤖 5. Плагины и специализации**

**Цель:** извлекать особые доменные сущности, например:

| **Плагин** | **Сущности** |
| --- | --- |
| medical.plugin | диагнозы, препараты, КЛД |
| legal.extractor | статьи, нормы, акты |
| finance.plugin | валюты, суммы, биржи |

* Стандартизировать интерфейс плагинов: initialize() и extract(text) → List[Entity].
* Разработать 2–3 примерных плагина.

**📈 6. Визуальная аналитика (BI/Power BI/Plotly Dash)**

**Цель:** использовать извлечённые данные в интерактивной аналитике.

**Идеи:**

* Dashboard активности по сущностям (временной график, география, частотность).
* Интеграция с PostgreSQL/ClickHouse для хранения результатов.
* Автоматическая генерация отчётов.

**✅ Следующий рекомендуемый шаг:**

**Создать модуль graph\_tools.py,** который:

* получает List[Entity],
* строит Graph (с тегами, контекстом, весами),
* экспортирует: .gexf, .json, .graphml, .html,
* может быть вызван как API или в пайплайне.

Хочешь — я сразу создам graph\_tools.py с базовыми функциями?

**Возможности для дальнейшего развития:**

1. **Интеллектуальный анализ**:
   * **Кластеризация сущностей**: Добавить функцию для обнаружения кластеров сущностей на основе их связей.
   * **Ранжирование сущностей**: Реализовать методы для ранжирования сущностей по их важности в графе (например, PageRank, Betweenness Centrality).
   * **Анализ связей**: Добавить функции для анализа связей между сущностями, например, поиск наиболее значимых связей.
2. **Геоинтеграция**:
   * **Геокодирование**: Добавить возможность связывать локации с географическими координатами и визуализировать их на карте.
   * **Экспорт в GeoJSON**: Реализовать экспорт графа в формат GeoJSON для использования в картографических приложениях.
3. **Интеграция с аналитикой**:
   * **Преобразование в pandas DataFrame**: Добавить функцию для преобразования графа в pandas DataFrame для дальнейшего анализа.
   * **Генерация отчётов**: Реализовать функции для генерации отчётов в формате PDF или HTML.
4. **Веб-интерфейс**:
   * **Интерактивная визуализация**: Добавить возможность интерактивной визуализации графа с помощью библиотек, таких как Plotly Dash или Cytoscape.js.
   * **API**: Реализовать REST API для загрузки сущностей и получения результатов анализа.
5. **Интеграция с визуализатором**:
   * **Передача графа в визуализатор**: Добавить возможность передачи графа в визуализатор для интерактивного анализа.
   * **Отображение отфильтрованных кластеров или маршрутов**: Реализовать функции для отображения отфильтрованных данных в визуализаторе.
6. **Генерация событий**:
   * **Мониторинг новых сущностей**: Добавить функцию для мониторинга новых сущностей или связей и отправки уведомлений.
   * **Алерты**: Реализовать систему алертов при появлении определённых связей или значимых узлов.

**📌 Пример реализации дополнительных функций:**

**1. Кластеризация сущностей:**

def detect\_communities(graph: nx.Graph) -> Dict[int, List[str]]:

"""

Обнаруживает кластеры сущностей в графе.

:param graph: Граф сущностей

:return: Словарь {номер кластера: [сущности]}

"""

import community as community\_louvain

partition = community\_louvain.best\_partition(graph)

clusters = defaultdict(list)

for node, group in partition.items():

clusters[group].append(node)

return dict(clusters)

**2. Ранжирование сущностей:**

def rank\_entities(graph: nx.Graph, method: str = "pagerank") -> Dict[str, float]:

"""

Ранжирует сущности в графе по их важности.

:param graph: Граф сущностей

:param method: Метод ранжирования ('pagerank', 'degree', 'betweenness')

:return: Словарь {сущность: важность}

"""

if method == "pagerank":

return nx.pagerank(graph)

elif method == "degree":

return dict(graph.degree())

elif method == "betweenness":

return nx.betweenness\_centrality(graph)

else:

raise ValueError("Unknown ranking method")

**3. Геокодирование:**

def geocode\_entities(graph: nx.Graph) -> nx.Graph:

"""

Добавляет географические координаты к локациям в графе.

:param graph: Граф сущностей

:return: Граф с добавленными координатами

"""

import geopy

geolocator = geopy.Nominatim(user\_agent="entity\_extractor")

for node in graph.nodes:

if graph.nodes[node]['label'] == 'LOCATION':

location = geolocator.geocode(node)

if location:

graph.nodes[node]['lat'] = location.latitude

graph.nodes[node]['lon'] = location.longitude

return graph

**4. Интерактивная визуализация:**

def visualize\_graph\_interactive(graph: nx.Graph, filename: str = "graph.html"):

"""

Визуализирует граф в интерактивном формате HTML.

:param graph: Граф сущностей

:param filename: Имя файла для сохранения

"""

import plotly.graph\_objects as go

import networkx as nx

pos = nx.spring\_layout(graph)

edge\_x = []

edge\_y = []

for edge in graph.edges():

x0, y0 = pos[edge[0]]

x1, y1 = pos[edge[1]]

edge\_x.extend([x0, x1, None])

edge\_y.extend([y0, y1, None])

edge\_trace = go.Scatter(

x=edge\_x, y=edge\_y,

line=dict(width=0.5, color='#888'),

hoverinfo='none',

mode='lines')

node\_x = []

node\_y = []

node\_text = []

for node in graph.nodes():

x, y = pos[node]

node\_x.append(x)

node\_y.append(y)

node\_text.append(node)

node\_trace = go.Scatter(

x=node\_x, y=node\_y,

mode='markers+text',

hoverinfo='text',

text=node\_text,

marker=dict(

showscale=True,

colorscale='YlGnBu',

reversescale=True,

color=[],

size=10,

colorbar=dict(

thickness=15,

title='Node Connections',

xanchor='left',

titleside='right'

),

line\_width=2))

fig = go.Figure(data=[edge\_trace, node\_trace],

layout=go.Layout(

title='Entity Graph',

titlefont\_size=16,

showlegend=False,

hovermode='closest',

margin=dict(b=20,l=5,r=5,t=40),

annotations=[ dict(

text="Entity Graph",

showarrow=False,

xref="paper", yref="paper",

x=0.005, y=-0.002 ) ],

xaxis=dict(showgrid=False, zeroline=False, showticklabels=False),

yaxis=dict(showgrid=False, zeroline=False, showticklabels=False)))

fig.write\_html(filename)

**📌 Заключение:**

Ваш модуль graph\_tools.py уже обладает мощным функционалом для работы с путями между сущностями. Добавление новых функций, таких как кластеризация, ранжирование и геокодирование, позволит сделать его ещё более универсальным и полезным для анализа данных. Если у вас возникнут вопросы или потребуется помощь в реализации новых функций, обращайтесь!

**Модуль для анализа путей между сущностями**

Вот полная реализация модуля graph\_tools.py для работы с путями между сущностями:

python

Copy

Download

*# 📄 Файл: graph\_tools.py*

*# 📂 Путь установки: librarian\_ai/core/graph\_tools.py*

import networkx as nx

import matplotlib.pyplot as plt

from typing import List, Dict, Optional

from entity\_extractor import Entity

from collections import defaultdict

def build\_entity\_graph(entities: List[Entity]) -> nx.Graph:

"""

Строит граф связей между сущностями на основе их контекста

:param entities: Список извлеченных сущностей

:return: Граф NetworkX с сущностями и связями

"""

G = nx.Graph()

*# Добавляем узлы с атрибутами*

for i, entity in enumerate(entities):

G.add\_node(

entity.text,

label=entity.label,

value=entity.value,

confidence=entity.confidence

)

*# Строим связи на основе общего контекста*

context\_map = defaultdict(list)

for entity in entities:

if entity.context:

context\_map[entity.context].append(entity.text)

for context, entity\_texts in context\_map.items():

for i in range(len(entity\_texts)):

for j in range(i+1, len(entity\_texts)):

G.add\_edge(

entity\_texts[i],

entity\_texts[j],

context=context,

weight=1.0 *# Можно настроить на основе confidence*

)

return G

def trace\_entity\_paths(

graph: nx.Graph,

source: str,

target: str,

algorithm: str = 'dijkstra'

) -> List[str]:

"""

Находит кратчайший путь между двумя сущностями в графе

:param graph: Граф сущностей

:param source: Исходная сущность

:param target: Целевая сущность

:param algorithm: Алгоритм поиска пути ('dijkstra', 'astar', 'bellman-ford')

:return: Список сущностей, представляющих кратчайший путь

"""

if source not in graph or target not in graph:

return []

try:

if algorithm == 'dijkstra':

return nx.shortest\_path(graph, source=source, target=target, weight='weight')

elif algorithm == 'astar':

return nx.astar\_path(graph, source=source, target=target, weight='weight')

elif algorithm == 'bellman-ford':

return nx.bellman\_ford\_path(graph, source=source, target=target, weight='weight')

else:

return nx.shortest\_path(graph, source=source, target=target)

except nx.NetworkXNoPath:

return []

def visualize\_path(

graph: nx.Graph,

path: List[str],

title: str = "Entity Path Visualization",

filename: Optional[str] = None

):

"""

Визуализирует найденный путь между сущностями

:param graph: Исходный граф

:param path: Список узлов пути

:param title: Заголовок графика

:param filename: Если указан, сохраняет в файл

"""

plt.figure(figsize=(12, 8))

*# Позиционирование узлов*

pos = nx.spring\_layout(graph, seed=42)

*# Отрисовка всего графа*

nx.draw\_networkx\_nodes(graph, pos, node\_size=300, node\_color='lightblue')

nx.draw\_networkx\_edges(graph, pos, alpha=0.2)

nx.draw\_networkx\_labels(graph, pos, font\_size=8)

*# Выделение пути*

if len(path) > 1:

path\_edges = list(zip(path[:-1], path[1:]))

nx.draw\_networkx\_nodes(graph, pos, nodelist=path, node\_size=500, node\_color='red')

nx.draw\_networkx\_edges(graph, pos, edgelist=path\_edges, width=2, edge\_color='red')

plt.title(title)

plt.axis('off')

if filename:

plt.savefig(filename, bbox\_inches='tight', dpi=300)

plt.close()

else:

plt.show()

def find\_all\_paths(

graph: nx.Graph,

source: str,

target: str,

cutoff: Optional[int] = None

) -> List[List[str]]:

"""

Находит все возможные пути между двумя сущностями

:param graph: Граф сущностей

:param source: Исходная сущность

:param target: Целевая сущность

:param cutoff: Максимальная длина пути

:return: Список всех возможных путей

"""

if source not in graph or target not in graph:

return []

try:

return list(nx.all\_simple\_paths(graph, source=source, target=target, cutoff=cutoff))

except nx.NetworkXNoPath:

return []

def calculate\_path\_strength(graph: nx.Graph, path: List[str]) -> float:

"""

Вычисляет силу пути на основе весов ребер

:param graph: Граф сущностей

:param path: Список узлов пути

:return: Общая сила пути (сумма весов)

"""

if len(path) < 2:

return 0.0

total\_strength = 0.0

for i in range(len(path)-1):

edge\_data = graph.get\_edge\_data(path[i], path[i+1])

total\_strength += edge\_data.get('weight', 1.0)

return total\_strength / (len(path)-1)

def get\_entity\_connections(graph: nx.Graph, entity: str, depth: int = 1) -> Dict[str, List[str]]:

"""

Возвращает все связи сущности до указанной глубины

:param graph: Граф сущностей

:param entity: Исходная сущность

:param depth: Глубина поиска связей

:return: Словарь {уровень: [сущности]}

"""

if entity not in graph:

return {}

connections = defaultdict(list)

visited = set()

queue = [(entity, 0)]

while queue:

current\_entity, current\_depth = queue.pop(0)

if current\_depth > depth:

continue

if current\_entity in visited:

continue

visited.add(current\_entity)

connections[current\_depth].append(current\_entity)

for neighbor in graph.neighbors(current\_entity):

if neighbor not in visited:

queue.append((neighbor, current\_depth + 1))

return dict(connections)

**Пример использования**

python

Copy

Download

from entity\_extractor import Entity

from graph\_tools import build\_entity\_graph, trace\_entity\_paths, visualize\_path

*# Пример списка сущностей*

entities = [

Entity(label="PERSON", text="Иван Иванов", value="Иван Иванов", context="Москва", confidence=0.9),

Entity(label="LOCATION", text="Москва", value="Москва", context="Россия", confidence=0.8),

Entity(label="ORGANIZATION", text="СберБанк", value="СберБанк", context="финансовый сектор", confidence=0.95),

Entity(label="PERSON", text="Петр Петров", value="Петр Петров", context="СберБанк", confidence=0.85),

Entity(label="EVENT", text="Встреча", value="Встреча", context="деловая встреча", confidence=0.85),

Entity(label="DATE", text="2023-09-15", value="2023-09-15", context="событие", confidence=0.9)

]

*# Построение графа*

graph = build\_entity\_graph(entities)

*# Поиск кратчайшего пути*

path = trace\_entity\_paths(graph, "Иван Иванов", "СберБанк")

print("Кратчайший путь:", path)

*# Визуализация пути*

visualize\_path(graph, path, title="Связь между Иван Иванов и СберБанк")

*# Поиск всех путей*

all\_paths = find\_all\_paths(graph, "Иван Иванов", "СберБанк", cutoff=3)

print("\nВсе возможные пути:")

for i, p in enumerate(all\_paths, 1):

print(f"{i}. {p} (сила: {calculate\_path\_strength(graph, p):.2f})")

*# Получение связей сущности*

connections = get\_entity\_connections(graph, "Иван Иванов", depth=2)

print("\nСвязи Иван Иванов:")

for level, ents in connections.items():

print(f"Уровень {level}: {ents}")

**Ключевые возможности**

1. **Построение графа сущностей**:
   * Автоматическое создание связей на основе общего контекста
   * Сохранение всех атрибутов сущностей
2. **Поиск путей**:
   * Кратчайший путь (несколько алгоритмов)
   * Все возможные пути между сущностями
   * Расчет силы/веса пути
3. **Анализ связей**:
   * Поиск всех связей сущности до указанной глубины
   * Визуализация связей
4. **Визуализация**:
   * Выделение найденных путей
   * Гибкие параметры отображения
   * Сохранение в файл

Модуль можно расширять, добавляя новые алгоритмы анализа графа и дополнительные визуализации.

**Модуль для анализа путей между сущностями**

Вот полная реализация модуля graph\_tools.py для работы с путями между сущностями:

python

Copy

Download

*# 📄 Файл: graph\_tools.py*

*# 📂 Путь установки: librarian\_ai/core/graph\_tools.py*

import networkx as nx

import matplotlib.pyplot as plt

from typing import List, Dict, Optional

from entity\_extractor import Entity

from collections import defaultdict

def build\_entity\_graph(entities: List[Entity]) -> nx.Graph:

"""

Строит граф связей между сущностями на основе их контекста

:param entities: Список извлеченных сущностей

:return: Граф NetworkX с сущностями и связями

"""

G = nx.Graph()

*# Добавляем узлы с атрибутами*

for i, entity in enumerate(entities):

G.add\_node(

entity.text,

label=entity.label,

value=entity.value,

confidence=entity.confidence

)

*# Строим связи на основе общего контекста*

context\_map = defaultdict(list)

for entity in entities:

if entity.context:

context\_map[entity.context].append(entity.text)

for context, entity\_texts in context\_map.items():

for i in range(len(entity\_texts)):

for j in range(i+1, len(entity\_texts)):

G.add\_edge(

entity\_texts[i],

entity\_texts[j],

context=context,

weight=1.0 *# Можно настроить на основе confidence*

)

return G

def trace\_entity\_paths(

graph: nx.Graph,

source: str,

target: str,

algorithm: str = 'dijkstra'

) -> List[str]:

"""

Находит кратчайший путь между двумя сущностями в графе

:param graph: Граф сущностей

:param source: Исходная сущность

:param target: Целевая сущность

:param algorithm: Алгоритм поиска пути ('dijkstra', 'astar', 'bellman-ford')

:return: Список сущностей, представляющих кратчайший путь

"""

if source not in graph or target not in graph:

return []

try:

if algorithm == 'dijkstra':

return nx.shortest\_path(graph, source=source, target=target, weight='weight')

elif algorithm == 'astar':

return nx.astar\_path(graph, source=source, target=target, weight='weight')

elif algorithm == 'bellman-ford':

return nx.bellman\_ford\_path(graph, source=source, target=target, weight='weight')

else:

return nx.shortest\_path(graph, source=source, target=target)

except nx.NetworkXNoPath:

return []

def visualize\_path(

graph: nx.Graph,

path: List[str],

title: str = "Entity Path Visualization",

filename: Optional[str] = None

):

"""

Визуализирует найденный путь между сущностями

:param graph: Исходный граф

:param path: Список узлов пути

:param title: Заголовок графика

:param filename: Если указан, сохраняет в файл

"""

plt.figure(figsize=(12, 8))

*# Позиционирование узлов*

pos = nx.spring\_layout(graph, seed=42)

*# Отрисовка всего графа*

nx.draw\_networkx\_nodes(graph, pos, node\_size=300, node\_color='lightblue')

nx.draw\_networkx\_edges(graph, pos, alpha=0.2)

nx.draw\_networkx\_labels(graph, pos, font\_size=8)

*# Выделение пути*

if len(path) > 1:

path\_edges = list(zip(path[:-1], path[1:]))

nx.draw\_networkx\_nodes(graph, pos, nodelist=path, node\_size=500, node\_color='red')

nx.draw\_networkx\_edges(graph, pos, edgelist=path\_edges, width=2, edge\_color='red')

plt.title(title)

plt.axis('off')

if filename:

plt.savefig(filename, bbox\_inches='tight', dpi=300)

plt.close()

else:

plt.show()

def find\_all\_paths(

graph: nx.Graph,

source: str,

target: str,

cutoff: Optional[int] = None

) -> List[List[str]]:

"""

Находит все возможные пути между двумя сущностями

:param graph: Граф сущностей

:param source: Исходная сущность

:param target: Целевая сущность

:param cutoff: Максимальная длина пути

:return: Список всех возможных путей

"""

if source not in graph or target not in graph:

return []

try:

return list(nx.all\_simple\_paths(graph, source=source, target=target, cutoff=cutoff))

except nx.NetworkXNoPath:

return []

def calculate\_path\_strength(graph: nx.Graph, path: List[str]) -> float:

"""

Вычисляет силу пути на основе весов ребер

:param graph: Граф сущностей

:param path: Список узлов пути

:return: Общая сила пути (сумма весов)

"""

if len(path) < 2:

return 0.0

total\_strength = 0.0

for i in range(len(path)-1):

edge\_data = graph.get\_edge\_data(path[i], path[i+1])

total\_strength += edge\_data.get('weight', 1.0)

return total\_strength / (len(path)-1)

def get\_entity\_connections(graph: nx.Graph, entity: str, depth: int = 1) -> Dict[str, List[str]]:

"""

Возвращает все связи сущности до указанной глубины

:param graph: Граф сущностей

:param entity: Исходная сущность

:param depth: Глубина поиска связей

:return: Словарь {уровень: [сущности]}

"""

if entity not in graph:

return {}

connections = defaultdict(list)

visited = set()

queue = [(entity, 0)]

while queue:

current\_entity, current\_depth = queue.pop(0)

if current\_depth > depth:

continue

if current\_entity in visited:

continue

visited.add(current\_entity)

connections[current\_depth].append(current\_entity)

for neighbor in graph.neighbors(current\_entity):

if neighbor not in visited:

queue.append((neighbor, current\_depth + 1))

return dict(connections)

**Пример использования**

python

Copy

Download

from entity\_extractor import Entity

from graph\_tools import build\_entity\_graph, trace\_entity\_paths, visualize\_path

*# Пример списка сущностей*

entities = [

Entity(label="PERSON", text="Иван Иванов", value="Иван Иванов", context="Москва", confidence=0.9),

Entity(label="LOCATION", text="Москва", value="Москва", context="Россия", confidence=0.8),

Entity(label="ORGANIZATION", text="СберБанк", value="СберБанк", context="финансовый сектор", confidence=0.95),

Entity(label="PERSON", text="Петр Петров", value="Петр Петров", context="СберБанк", confidence=0.85),

Entity(label="EVENT", text="Встреча", value="Встреча", context="деловая встреча", confidence=0.85),

Entity(label="DATE", text="2023-09-15", value="2023-09-15", context="событие", confidence=0.9)

]

*# Построение графа*

graph = build\_entity\_graph(entities)

*# Поиск кратчайшего пути*

path = trace\_entity\_paths(graph, "Иван Иванов", "СберБанк")

print("Кратчайший путь:", path)

*# Визуализация пути*

visualize\_path(graph, path, title="Связь между Иван Иванов и СберБанк")

*# Поиск всех путей*

all\_paths = find\_all\_paths(graph, "Иван Иванов", "СберБанк", cutoff=3)

print("\nВсе возможные пути:")

for i, p in enumerate(all\_paths, 1):

print(f"{i}. {p} (сила: {calculate\_path\_strength(graph, p):.2f})")

*# Получение связей сущности*

connections = get\_entity\_connections(graph, "Иван Иванов", depth=2)

print("\nСвязи Иван Иванов:")

for level, ents in connections.items():

print(f"Уровень {level}: {ents}")

**Ключевые возможности**

1. **Построение графа сущностей**:
   * Автоматическое создание связей на основе общего контекста
   * Сохранение всех атрибутов сущностей
2. **Поиск путей**:
   * Кратчайший путь (несколько алгоритмов)
   * Все возможные пути между сущностями
   * Расчет силы/веса пути
3. **Анализ связей**:
   * Поиск всех связей сущности до указанной глубины
   * Визуализация связей
4. **Визуализация**:
   * Выделение найденных путей
   * Гибкие параметры отображения
   * Сохранение в файл

Модуль можно расширять, добавляя новые алгоритмы анализа графа и дополнительные визуалОтлично, давай проведём обзор текущего состояния **Librarian AI** — интеллектуальной системы для извлечения, анализа и визуализации сущностей — и определим план дальнейших шагов.

**✅ Что уже сделано (ядро системы):**

**📦 Базовые модули**

| **Модуль** | **Статус** | **Описание** |
| --- | --- | --- |
| entity\_extractor.py | ✅ Завершён | Извлечение сущностей из текста (Natasha, spaCy, regex). Поддержка плагинов, YAML-конфигов, словарей, кэша. |
| entity\_extractor\_advanced.py | ✅ Завершён | Расширенная версия с поддержкой плагинов, пользовательских словарей, конфигурации, телеметрии, кэширования. |
| visualizer.py | ✅ Завершён | Профессиональная визуализация графа сущностей с Plotly/Matplotlib, экспорт в PNG, GEXF и интерактив. |
| graph\_tools.py | ✅ Базовая версия готова | Построение, экспорт, фильтрация, резюме, поиск путей. Планируется развитие. |

**📂 Структура проекта**

* librarian\_ai/core/entity\_extractor\_advanced.py — основа логики
* librarian\_ai/core/visualizer.py — визуализация
* librarian\_ai/core/graph\_tools.py — графовый анализ
* config.yaml / json — конфигурации
* 🔄 Поддержка CLI и расширения через плагины

**🚀 Что предстоит сделать (дорожная карта)**

**🔍 1. Анализ и интеллект**

* Кластеризация узлов (detect\_communities)
* Ранжирование узлов по важности (rank\_entities)
* Выявление скрытых паттернов в графе

**🧩 2. Интеграции и расширения**

* Подключение внешних баз (Wikidata, DBpedia, GeoNames)
* Интеграция с GPT-моделями (explain\_link, generate\_description)
* Связь с корпоративными словарями, документами (1С, PDF, базы данных)

**📊 3. Отчётность и аналитика**

* Экспорт в HTML и PDF отчёты
* Pandas DataFrame из графа
* Модуль анализа сущностей по категориям (частота, важность, аномалии)

**🌐 4. Web-интерфейс и API**

* FastAPI REST-интерфейс:
  + POST /extract
  + POST /graph
  + GET /visualize
* Streamlit или Dash визуализатор (upload text → see graph)
* Микросервисная упаковка (Docker, OpenAPI)

**🧠 5. Обогащение знаний**

* Хранилище графов (MongoDB + Neo4j + FAISS/Weaviate)
* Автоматическое связывание новых текстов с уже существующими сущностями
* Обратная связь от пользователя (релевантность, соединения)

**📌 Прикладные сценарии использования Librarian AI:**

| **Сценарий** | **Состояние** |
| --- | --- |
| 💼 Анализ юридических документов | 🟢 реализуется |
| 🏥 Медицинские карты и термины | 🟡 частично |
| 📚 Классификация и навигация по статьям | 🟢 |
| 🔎 Распознавание персон, организаций, событий | 🟢 |
| 📍 Геоанализ и маршруты по контексту | 🔜 |

**🔧 Технические предложения:**

* Вынести конфигурации и плагины в отдельные модули: config\_loader.py, plugin\_manager.py
* Подключить pytest и coverage для юнит-тестов
* Реализовать librarian\_cli.py для запуска в терминале
* Объединить всё в модульный пакет (setup.py, \_\_init\_\_.py)

**🤖 Системная архитектура (целевая)**

css

КопироватьРедактировать

📄 Текст/документ

↓

[ EntityExtractor ]

↓

[ Граф сущностей ] ←→ [ Расширения / плагины ]

↓ ↑

[ GraphTools ] [ ML / GPT ]

↓

[ Visualizer / Reports / API ]

**🔜 Предлагаю следующий шаг:**

1. Выбрать: rank\_entities() или detect\_communities()
2. Параллельно: оформить CLI-интерфейс librarian\_cli.py
3. И начать упаковывать модуль в pip-дистрибутив

Хочешь — начнём с любого из этих направлений. Что выбираем?

изации.

🚀 Отлично, мы продолжаем развитие ядра **Librarian AI**. Вот текущий **статус системы** и **следующие ключевые шаги**:

**✅ Что уже сделано:**

**🧠 1. Ядро извлечения сущностей (entity\_extractor\_advanced.py)**

* Поддержка **русского, английского, немецкого, французского** языков.
* Конфигурация через **YAML/JSON**.
* Механизм **нормализации** дат, чисел, имён, организаций.
* **Пользовательские словари**, **регулярные выражения**, **плагины**.
* **Кэширование**, **многопроцессорность**, **телеметрия**.
* API: extract\_entities, batch\_extract, get\_telemetry, reset\_stats.

**🔎 2. Модуль визуализации (visualizer.py)**

* Поддержка **расширенной визуализации** с matplotlib и plotly.
* Экспорт в GEXF, PNG, SVG, PDF.
* Конфигурируемый класс VisualizationConfig.
* Раскладки графов (spring, circular, shell, kamada\_kawai).

**🌐 3. Работа с графами (graph\_tools.py)**

* Построение графа из сущностей.
* Экспорт/импорт в **GraphML, GEXF, JSON, Pickle**.
* get\_graph\_summary(), filter\_graph(), trace\_entity\_paths(), rank\_entities().

**📍 Что предстоит:**

**🧩 4. Добавить функции анализа:**

* detect\_communities(graph) — кластеризация сущностей (Louvain, Girvan–Newman).
* annotate\_graph\_with\_weights() — добавить веса на связи (например, частота контекстов).
* expand\_graph\_from\_knowledge\_base() — подключение внешних знаний (Wikidata, DBpedia).

**📄 5. Генерация отчетов:**

* Автоматическое создание **PDF/HTML отчётов** с графиками и описаниями.
* Генерация Word-документов с вложениями (через python-docx/reportlab).

**⚙️ 6. API-интерфейс:**

* REST API для интеграции с внешними сервисами (FastAPI или Flask).
* POST /extract, GET /summary, POST /visualize.

**🧱 7. Модули интеграции:**

* 📁 Поддержка обработки **файлов**: .docx, .pdf, .eml, .html, .txt.
* 🔄 Интерфейс импорта из папки (режим «наблюдателя»).
* 🤖 Интеграция с AI: генерация вопросов по графу, чат-бот.

**📊 8. Связь с аналитикой:**

* Преобразование графов в **pandas.DataFrame**.
* Вывод статистик в виде таблиц, heatmap, top-N.

**🎯 Приоритет на сейчас**

Предлагаю следующий пошаговый план:

| **Шаг** | **Задача** | **Пример функции** |
| --- | --- | --- |
| 1 | **Кластеризация** | detect\_communities() |
| 2 | **PDF/HTML отчёт по графу** | generate\_report() |
| 3 | **API-интерфейс FastAPI** | POST /extract |
| 4 | **Импорт документов из файлов** | load\_from\_file() |
| 5 | **Подключение Wikidata (RDF)** | expand\_graph() |

Хочешь начать с кластеризации (detect\_communities) или с отчётности (PDF/HTML)? Или пойдём по другому маршруту?