**Структура архитектурного документа Python-системы обработки архивных документов для 1С для ИП**

## 1. 📌 Введение

### 1.1. ****Назначение и аудитория документа****

Документ предназначен для бизнес-аналитиков, IT-архитекторов, системных администраторов и специалистов по автоматизации, отвечающих за внедрение, поддержку и развитие системы автоматизированной обработки архивных документов и интеграции с 1С.

* **Аудитория:**
  + Проектная команда внедрения
  + Разработчики и интеграторы
  + Ответственные сотрудники бизнес-подразделений
  + Внешние аудиторы, заказчик

### 1.2. ****Общая миссия, задачи и зона применения системы****

* **Миссия:**  
  — Исключить человеческий фактор и рутину в обработке, структурировании и загрузке архивных документов, гарантировать полную прозрачность и управляемость бизнес-процесса.
* **Задачи:**
  + Автоматизировать поиск, фильтрацию, разбор и анализ документов (PDF, DOCX) в сложных иерархиях папок (по кредиторам и должникам)
  + Стандартизировать и структурировать ключевые поля для загрузки в 1С и другие корпоративные системы
  + Исключить повторную обработку одних и тех же файлов за счёт строгого контроля и ведения журналов обработки
  + Повысить качество данных через шаблоны, справочники и автоматическую валидацию
  + Обеспечить лёгкую масштабируемость, обновляемость и развитие системы (по принципам Кайзен)

### 1.3. ****Ожидаемые результаты внедрения****

* Существенное снижение числа ошибок при подготовке и загрузке документов
* Резкое сокращение времени на обработку (от человека — к машине)
* Прозрачные и единые стандарты хранения, журналирования и аудита процессов
* Минимизация ручных вмешательств и рисков “человеческого фактора”

| **№** | **Раздел** | **Стр.** |
| --- | --- | --- |
| **1** | **📌 Введение** | 1 |
|  | 1.1 Назначение и аудитория документа |  |
|  | 1.2 Миссия, задачи и зона применения |  |
|  | 1.3 Ожидаемые результаты и принципы построения |  |
| 2 | **🗺️ Общая схема бизнес-процесса** | 4 |
|  | 2.1. Краткое описание бизнес-процесса |  |
|  | 2.2. Схема потоков данных |  |
|  | **2.3. Чек-лист ответственных на каждом этапе** |  |
|  | **2.4. Основные точки контроля и принятия решений** |  |
| 3 | **⚙️ Подготовка окружения** | 6 |
|  | **3.1. Минимальные системные требования** |  |
|  | **3.2. Создание и активация виртуального окружения** |  |
|  | **3.3. Установка системных зависимостей**(Linux/Windows) |  |
|  | **3.4. Установка Python-зависимостей** |  |
|  | **3.5. Копирование/подготовка справочников** |  |
|  | **3.6. Автозагрузка и управление скриптом** |  |
|  | **3.7. Проверка запуска и тестовая диагностика** |  |
| 4 | 🧩 Архитектура проекта и модулей | 9 |
|  | 4.1. Логическая структура папок и файлов |  |
|  | 4.2. Основные модули и их назначение |  |
|  | 4.3. Рекомендации по структуре |  |
|  | 4.4. Итог и масштабируемость |  |
| 5 | **📥 Механика загрузки и проверки справочников** | 11 |
|  | 5.1. Где и как хранятся справочники |  |
|  | 5.2. Проверка справочников при запуске |  |
|  | 5.3. Механика валидации и авто-логирования |  |
|  | 5.4. Резервное копирование и защита от потери |  |
|  | 5.5. Формат журналирования процесса подготовки |  |
|  | 5.6. Форматы справочников |  |
|  | 5.7. Связь справочников и скриптов |  |
| 6 | **📁 Итерация по архиву** | 19 |
|  | 6.1. Алгоритм обхода кредиторов и должников |  |
|  | 6.2. Валидация структуры и фиксация ошибок |  |
|  | 6.3. Модули, отвечающие за обход и контроль структуры |  |
|  | 6.4. Результаты этапа итерации |  |
| 7 | **📝 Парсинг и анализ** | 21 |
|  | 7.1. Алгоритм фильтрации и проверки форматов |  |
|  | 7.2. Извлечение и интерпретация текста |  |
|  | 7.2.1. Алгоритм выбора метода извлечения (PDF, DOCX, XLSX и др.) |  |
|  | 7.2.2. Сценарии по форматам |  |
|  | 7.2.3. Унификация результатов |  |
|  | 7.2.4. Примеры кода для каждого формата |  |
|  | 7.2.5. Обработка ошибок и логирование |  |
|  | 7.3. Выделение ключевых полей |  |
|  | 7.3.1. Стратегия: регулярки, шаблоны, справочники |  |
|  | 7.3.2. Порядок и приоритеты извлечения |  |
|  | 7.3.3. Обработка неоднозначных случаев |  |
|  | 7.4. Интеграция с AI |  |
|  | 7.4.1. Структура запроса и отправка текста |  |
|  | 7.4.2. Обработка и валидация ответа |  |
|  | 7.4.3. Используемый промт и логика генерации |  |
|  | 7.4.4. Типовые ошибки и fallback-механизм |  |
|  | 7.5. Проверка полноты и поиск дублей |  |
|  | 7.5.1. Алгоритм поиска дублей |  |
|  | 7.5.2. Критерии неполных данных |  |
|  | 7.5.3. Примеры и фиксация ошибок |  |
|  | 7.6. Примеры парсинга и результата |  |
| 8 | **🔄 Загрузка в 1С** | 55 |
|  | 8.1. Формат данных для API 1С |  |
|  | 8.2. Алгоритм автоматической загрузки |  |
|  | 8.3. Контроль и прозрачность |  |
| 9 | 🗂️ **Логирование и аудит** | 61 |
|  | 9.1. Ведение журналов обработки и ошибок |  |
|  | 9.2. Формат и хранение логов |  |
|  | 9.3. Доступность и аудит |  |
| 10 | ♻️ **Запуск, координация и “сквозной сценарий”** | 63 |
|  | 10.1. Общая логика запуска |  |
|  | 10.2. Порядок запуска и связей между модулями |  |
|  | 10.3. Логические зависимости и события |  |
|  | 10.4. Чек-лист готовности сквозного запуска |  |
|  | 10.5. Сценарии запуска |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# 2. 🗺️ ****Общая схема бизнес-процесса****

## **2.1. Краткое описание бизнес-процесса**

Автоматизированный цикл проходит через следующие основные этапы:

1. **Подготовка к обработке**  
   — Проверка и загрузка всех справочников, журналов и файлов контроля.  
   — Сбор финального списка “кредитор-должник” для обработки (без дублей, с учётом уже обработанных кейсов).
2. **Итерация и обход архива**  
   — Для каждого кредитора и должника рекурсивный проход по всем папкам и подпапкам.  
   — Сбор всех файлов, фильтрация по форматам, фиксация ошибок доступа.
3. **Фильтрация и обработка файлов**  
   — Исключение неразрешённых форматов, журналирование ошибок и “мусора”.  
   — Парсинг валидных файлов на нужные поля, обработка вложенных структур.
4. **Извлечение и стандартизация данных**  
   — Определение типа документа (входящий/исходящий), поиск дат, ID, адресата и прочих ключевых полей.  
   — Использование AI для коротких смысловых выжимок (“прошу…”, комментарий).  
   — Проверка полноты, анализ дублей, подготовка к формированию выгрузки.
5. **Формирование выгрузки**  
   — Создание строки-результата (Excel/JSON) для каждого кейса.  
   — Пометка флагов “неполные данные” и других статусов.  
   — Формирование финального файла выгрузки с уникальным именем.
6. **Загрузка в 1С**  
   — Приведение всех значений к формату 1С (справочники, дата, ID).  
   — Загрузка вручную или через API, контроль результата.  
   — Повторная загрузка для строк с ошибками, отчёт о загруженных данных.
7. **Журналирование, аудит, доработка**  
   — Ведение логов ошибок, инцидентов, сложных/неразобранных файлов.  
   — Регулярный разбор нештатных случаев ответственным специалистом.
8. **Поддержка, развитие, “Кайзен”**  
   — Постоянное обновление справочников, шаблонов, форматов.  
   — Системный разбор ошибок и предложений по улучшению, учёт новых сценариев.

## **2.2. Схема потоков данных**

graph TD

A[Старт процесса] --> B{Проверка справочников и журналов}

B -->|ОК| C[Сбор финального списка кредиторов/должников]

B -->|Ошибка| Z[Остановка, уведомление ответственного]

C --> D{Итерация по папкам}

D --> E[Фильтрация и обработка файлов]

E --> F[Извлечение полей и стандартизация]

F --> G[Формирование выгрузки (Excel/JSON)]

G --> H{Загрузка в 1С}

H -->|Успех| I[Фиксация результата, журналирование]

H -->|Ошибка| J[Журнал ошибок, повторная загрузка]

I --> K[Анализ, доработка, поддержка]

J --> K

K --> A

## **2.3. Чек-лист ответственных на каждом этапе**

* **Администратор**:  
  — Следит за целостностью справочников, структурой папок, логами.
* **Оператор/парсер**:  
  — Запускает процесс, реагирует на ошибки, контролирует обработку файлов, парсинг.
* **Аналитик**:  
  — Проверяет неразобранные и проблемные случаи, обновляет шаблоны и справочники, ведёт статистику по ошибкам.

## **2.4. Основные точки контроля и принятия решений**

* Каждый шаг процесса завершается чек-листом:  
  — Всё ли загружено? Нет ли ошибок доступа?  
  — Нет ли дублей? Всё ли в нужном формате?  
  — Всё ли выгружено? Есть ли файлы “на доработку”?  
  — Обновлены ли справочники по итогам разбора?

# 3. ⚙️ ****Установка зависимостей и подготовка переменного окружения****

### ****3.1. Минимальные системные требования****

| **Компонент** | **Минимальная конфигурация** |
| --- | --- |
| Процессор (CPU) | 4 ядра, 2.0+ ГГц (современный i5/i7, Ryzen, Xeon — на выбор) |
| Оперативная память | 8–16 ГБ (достаточно для обычного документооборота) |
| SSD/Диск | 50 ГБ+ под систему и рабочие файлы, 100 ГБ+ под архивы/логи |
| ОС | Ubuntu 22.04+ / Debian 11+ / Windows 10/Server 2019+ |
| Python | 3.10+ (LTS-версия) |
| Интернет | Стабильное подключение (для работы с API Яндекса/внешними сервисами) |

**Рекомендуется запуск в виртуальном окружении Python (venv/anaconda)**  
Это обеспечит изоляцию и лёгкий перенос на другой ПК/сервер.

### ****3.2. Создание и активация виртуального окружения****

python3 -m venv /path/to/project/venv

source /path/to/project/venv/bin/activate

### ****3.3. Установка системных зависимостей****(Linux/Windows)

Для **Ubuntu/Debian**:

sudo apt update

sudo apt install -y \

python3 python3-pip unzip ffmpeg git htop curl \

poppler-utils tesseract-ocr libtesseract-dev \

libmagic1 libxml2 libxslt1.1 \

libreoffice  
  
  
  
 **poppler-utils** — для извлечения текста из PDF (pdf2text, pdfimages и т.д.)

 **tesseract-ocr** — для OCR (распознавания текста на изображениях в PDF/JPEG/PNG)

 **libmagic1** — для определения формата файла “на лету”

 **libreoffice** — для конвертации DOC, DOCX, XLSX → PDF/TXT

### ****3.4. Установка Python-зависимостей****

pip install -U pip

pip install \

pandas \

openpyxl \

loguru \

pyyaml \

requests \

tqdm \

fastapi \

uvicorn \

aiofiles \

python-docx \

python-pptx \

PyPDF2 \

pdfminer.six \

pillow \

pytesseract \

python-magic \

python-multipart \

textract \

pdf2image

# Для работы с AI (если требуется)

pip install yandexcloud

**Пояснения:**

* **PyPDF2, pdfminer.six** — обработка PDF, извлечение текста и метаданных
* **pytesseract, pillow, pdf2image** — OCR для сканов и картинок внутри PDF/JPG/PNG
* **python-docx, python-pptx** — обработка Word/PowerPoint, вытаскивание текста и вложений
* **python-magic** — определение “честного” формата файла
* **textract** — универсальный “комбайн” для сложных случаев (поддерживает разные типы файлов)

### ****3.5. Копирование/подготовка справочников****

* Поместить все актуальные справочники (formats.csv, adresat.csv, responsible.csv, events.csv, validators.yaml, creditors\_to\_process.csv) в папку /configs/
* Создать (если нет) логи /logs/process\_log.csv, /logs/error\_log.csv

### ****3.6. Автозагрузка и управление скриптом****

* Рекомендовано запускать основной скрипт (main.py) через systemd (Linux) или Task Scheduler (Windows) с правами обычного пользователя.
* Для простого включения/отключения — создать .bat/.sh скрипт или ярлык на рабочем столе для запуска/остановки.
* Если нужна “безопасная пауза” — реализовать флаг “pause” (например, наличие файла /pause.flag временно останавливает работу скрипта).

### ****3.7. Проверка запуска и тестовая диагностика****

source /path/to/project/venv/bin/activate

python main.py --check

# Проверяется:

# - Чтение справочников

# - Доступность API Яндекс (по тестовому ключу)

# - Доступность логов/каталогов

# - Нет критических ошибок

* Проверить логи /logs/ — ошибок не должно быть.
* Проверить папки /data/, /exports/ — должны быть доступны на запись и чтение.

### ****✅ На этом этапе:****

* Система готова к работе с внешним API (Яндекс и др.).
* Нет необходимости в мощных GPU или избыточной памяти: достаточно стандартного серверного или офисного ПК.
* Проект развёрнут в виртуальном окружении, легко переносится и обслуживается.
* Запуск/остановка легко доступны для любого пользователя — как по расписанию, так и вручную.

# ****4. 📂 Структура проекта****

## **4.1. Логическая структура папок и файлов**

Для максимальной прозрачности, поддержки и масштабируемости проекта структура должна быть следующей:

/project-root/

│

├── app/ # Все основные Python-модули логики

│ ├── main.py # Главная точка входа, управляет всем процессом

│ ├── config.py # Загрузка настроек и переменных окружения

│ ├── utils.py # Вспомогательные функции (даты, проверки и др.)

│ ├── filewalker.py # Рекурсивный обход структуры архивов/папок

│ ├── validator.py # Проверка валидности полей и структуры данных

│ ├── parser.py # Извлечение и нормализация данных из документов

│ ├── ai\_client.py # Интеграция с внешним AI (API Яндекса и пр.)

│ ├── exporter.py # Формирование итоговых JSON-выгрузок/отчётов

│ ├── state\_manager.py # Ведение журналов процесса и ошибок (JSON)

│ └── logs/ # (Внутренние модули для логирования, если требуется)

│

├── configs/ # Только справочники для сотрудников (только CSV/YAML)

│ ├── formats.csv # Разрешённые расширения файлов

│ ├── adresat.csv # Справочник адресатов

│ ├── responsible.csv # Справочник ответственных

│ ├── events.csv # Справочник типов мероприятий

│ ├── validators.yaml # Шаблоны и правила валидации (если нужны сложные проверки)

│ ├── creditors\_to\_process.csv # Список кредиторов и статусы (обрабатывается вручную)

│ └── config.yaml # Глобальные параметры, если используются

│

├── data/ # Архив документов (файловая структура кредиторов/должников)

│ └── ... # Неограниченная глубина и вложенность

│

├── exports/ # Результаты обработки (только JSON)

│ ├── выгрузка\_YYYYMMDD\_N.json # Итоговые выгрузки для 1С/архива

│ └── ... # Каждый новый поток — отдельный файл

│

├── logs/ # Только журналы и логи в JSON

│ ├── process\_log.json # Статус и детали обработки файлов

│ ├── error\_log.json # Лог ошибок (все причины, сообщения, дата/время)

│ ├── duplicates\_log.json # Найденные дубли (номер ИП, дата, путь)

│ ├── not\_processed.json # Необработанные и невалидные документы

│ └── pause.flag # Файл-пауза (для ручной остановки)

│

├── venv/ # Виртуальное окружение Python (обязательно)

│

├── README.md # Инструкция и базовое описание работы

├── requirements.txt # Список зависимостей Python

└── setup.py # (опционально) для пакетирования, если потребуется

**4.2. Основные модули и их назначение**

| **Модуль / Скрипт** | **Описание** |
| --- | --- |
| main.py | Главная точка входа, управляет всем процессом и обработкой ошибок |
| config.py | Загрузка настроек, обработка переменных окружения |
| utils.py | Вспомогательные функции (даты, проверки, сериализация, пр.) |
| filewalker.py | Рекурсивный обход архивов/папок, фильтрация файлов |
| validator.py | Проверка валидности полей и структуры данных |
| parser.py | Извлечение и нормализация данных из документов |
| ai\_client.py | Интеграция с внешним AI (API Яндекса и др.), обработка промптов |
| exporter.py | Формирование итоговых JSON-выгрузок, экспорт в Excel (при необходимости) |
| state\_manager.py | Ведение всех журналов (process\_log, errors, дубли, неразобранные) |
| logs/ | Модули для логирования и ротации (если требуется) |
| README.md | Краткая инструкция и описание проекта |
| requirements.txt | Список всех Python-зависимостей |

**4.3. Рекомендации по структуре**

* **Все справочники** — только в /configs/, чтобы их можно было редактировать и быстро аудитить.
* **Все журналы и логи** — только в /logs/, выгрузки — в /exports/.
* **Модули независимы:** можно менять API или экспорт без риска сломать парсер или валидатор.
* **Работа только из-под venv:** никаких глобальных зависимостей.
* **Вся документация и инструкции — в README.md.**

**Итог**

* Структура проста, модульна, интуитивно понятна.
* Любой новый разработчик или админ быстро включится в работу.
* Легко масштабируется и дорабатывается по мере роста требований.

# ****5. 📥 Механика загрузки и проверки справочников****

## **5.1. Где и как хранятся справочники**

* **Все справочники (CSV, YAML)** — всегда в папке /configs/.
* **Файлы, которые редактируют сотрудники:**
  + formats.csv — допустимые расширения (PDF, DOCX…)
  + adresat.csv — справочник адресатов (ФССП, банки…)
  + responsible.csv — справочник фамилий и ответственных
  + events.csv — типы мероприятий
  + creditors\_to\_process.csv — список кредиторов и ссылок для отбора (ключевой для запуска цикла)
* **Файлы для сложной валидации:**
  + validators.yaml — регулярные выражения, шаблоны, if/then-правила

## **5.2. Проверка справочников при запуске**

**Алгоритм:**

1. **Перед стартом основного цикла** (main.py) все справочники проверяются на:
   * Доступность (существует, не пустой)
   * Читаемость (валидная кодировка, корректные разделители)
   * Логичность и полноту (нет дубликатов, пустых строк, для каждого кредитора есть статус)
   * Отсутствие критических ошибок формата (не “битый” CSV, не нарушена структура)
2. **Если справочник не найден/битый:**
   * Процесс не стартует, ошибка записывается в error\_log.json
   * Ответственный получает уведомление (если автоматизация — e-mail/внутреннее сообщение)

### ****Взаимодействие с кодом****

* **Модули, которые работают с этими файлами:**
  + config.py — функции загрузки файлов (load\_csv, load\_yaml), хранение в глобальных переменных или DI-контейнере.
  + validator.py — проверки структуры, уникальности, полноты.
  + state\_manager.py — логирование статусов загрузки, автоматическое создание бэкапов, запись ошибок.

#### **Пример кода для загрузки и проверки**

# config.py

import pandas as pd

import yaml

from pathlib import Path

CONFIG\_DIR = Path(\_\_file\_\_).parent.parent / "configs"

def load\_csv(filename):

path = CONFIG\_DIR / filename

try:

df = pd.read\_csv(path, encoding="utf-8")

if df.empty:

raise ValueError("Пустой справочник")

return df

except Exception as e:

log\_error(stage="load\_config", file=filename, error=str(e), critical=True)

raise

def load\_yaml(filename):

path = CONFIG\_DIR / filename

try:

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

data = yaml.safe\_load(f)

return data

except Exception as e:

log\_error(stage="load\_config", file=filename, error=str(e), critical=True)

raise

# validator.py

def validate\_formats(df):

assert "extension" in df.columns, "Нет поля extension"

# все расширения должны быть строчными

if not all(df["extension"].str.islower()):

log\_error(stage="validate", file="formats.csv", error="Extension не в нижнем регистре", critical=True)

return False

return True

## **5.3. Механика валидации и авто-логирования**

* При любой ошибке в справочнике (ошибка чтения, пустой файл, дубликаты, невалидные значения):
  + В лог (error\_log.json) пишется блок:
  + {
  + "datetime": "2024-06-20T11:05:00",
  + "stage": "load\_config",
  + "file": "adresat.csv",
  + "error": "Duplicate entries in field 'address'",
  + "critical": true
  + }
* Если все справочники загружены успешно — в process\_log.json фиксируется успех и все параметры запуска.
* **Все дальнейшие проверки (валидность ссылок, наличие статуса “к обработке” и т.д.) — тоже фиксируются, ошибки валидируются до начала итерации по архиву.**

## **5.4. Резервное копирование и защита от потери**

* Перед любым автоматическим изменением справочника (если реализовано в будущем) —  
  файл автоматически копируется в /configs/backups/ с датой и временем.
* Операторы всегда знают:  
  “Если что-то пошло не так, можно откатиться на любой предыдущий вариант”.

## **5.5. Формат журналирования процесса подготовки**

* Для каждого запуска системы создаётся запись о статусе загрузки каждого справочника:
* {
* "datetime": "2024-06-20T10:59:00",
* "event": "configs\_check",
* "results": {
* "formats.csv": "OK",
* "adresat.csv": "OK",
* "responsible.csv": "OK",
* "events.csv": "OK",
* "validators.yaml": "OK",
* "creditors\_to\_process.csv": "OK"
* }
* }
* Все критические сбои мгновенно фиксируются и видны ответственному лицу.

## **5.6. Форматы справочников**

#### **formats.csv**

Справочник допустимых расширений файлов для обработки. Хранится в /configs/.

| **Поле** | **Описание** | **Пример** |
| --- | --- | --- |
| extension | Расширение файла (строчно) | pdf |
| description | Краткое описание (опционально) | Документ PDF |
| parser | Название парсера | pdf, docx, xlsx |

**Пример:**

extension,description,parser

pdf,Документ PDF,pdf

docx,Документ Word,docx

doc,Старый формат Word,doc

xlsx,Таблица Excel,xlsx

**adresat.csv**

*Справочник адресатов (куда могут быть адресованы документы).  
Файл хранится в /configs/.*

| **Поле** | **Описание** | **Пример** |
| --- | --- | --- |
| name | Название организации | ФССП |
| code | Код для 1С/внутренний | FSSP |
| comments | Примечание (опционально) | Федеральная служба судебных приставов |

**Пример содержимого:**

name,code,comments

ФССП,FSSP,Федеральная служба судебных приставов

Сбербанк,SBRF,Крупнейший банк

ВТБ,VTB,Второй по активам банк РФ

**responsible.csv**

*Справочник сотрудников/ответственных лиц для поиска в документах.  
Файл хранится в /configs/.*

| **Поле** | **Описание** | **Пример** |
| --- | --- | --- |
| surname | Фамилия | Иванов |
| name | Имя | Иван |
| patronymic | Отчество (опционально) | Иванович |
| position | Должность (опционально) | Юрист |

**Пример содержимого:**

surname,name,patronymic,position

Иванов,Иван,Иванович,Юрист

Петров,Пётр,Петрович,Оператор

Сидоров,Сидор,,Менеджер

**events.csv**

*Справочник типов мероприятий.  
Файл хранится в /configs/.*

| **Поле** | **Описание** | **Пример** |
| --- | --- | --- |
| event | Тип мероприятия | Жалоба |
| code | Код (опционально) | ZHALOBA |
| comments | Примечание (опционально) | Документ на жалобу |

**Пример содержимого:**

event,code,comments

Жалоба,ZHALOBA,Документ на жалобу

Заявление,STATEMENT,Общее заявление

Отзыв,RECALL,Обращение по отзыву

**Используется для:**

* Нормализации поля “Тип мероприятия” при парсинге и формировании выгрузки.
* Сопоставления с полями 1С при отправке по API.
* Валидации новых типов: если найден новый тип — логируется и выносится на добавление.

**Ошибки (дубли, пустые поля, неиспользуемые коды) — фиксируются в error\_log.json.**

**creditors\_to\_process.csv**

*Справочник кредиторов для обработки.  
Файл хранится в /configs/.  
Редактируется сотрудниками.*

| **Поле** | **Описание** | **Пример** |
| --- | --- | --- |
| creditor | Название кредитора | Сбербанк |
| link | Сетевая/локальная ссылка на папку | \asf01\files\Сбербанк |
| status | Статус (“к обработке”, “пропустить”) | к обработке |
| comments | Примечание (опционально) | Только новые дела |

**Пример содержимого:**

creditor,link,status,comments

Сбербанк,\\asf01\files\Сбербанк,к обработке,

Почта Банк,\\asf01\files\ПочтаБанк,пропустить,Не обрабатывать до уточнения

**Используется для:**

* Определения, какие папки-кредиторов будут сканироваться и включены в цикл обработки.
* Исключения ненужных/замороженных кредиторов из процесса.
* Сопоставления ссылок при обходе файловой системы.
* Проверки корректности структуры (обязательно: уникальность названий, наличие статуса).

**Ошибки (дубли, пустые значения, битые ссылки) — фиксируются в error\_log.json, обработка таких записей невозможна.**

**validators.yaml**

*Файл сложных правил валидации полей.  
Хранится в /configs/.  
Необходим для проверки форматов дат, номеров, типов мероприятий и пр.  
Редактируется только ответственным IT/аналитиком.*

**Пример структуры:**

date:

regex: '^(\d{2})\.(\d{2})\.(\d{4})$'

description: 'Дата в формате ДД.ММ.ГГГГ'

number\_ip:

regex: '^\d{6}/\d{2}/\d{5}-ИП$'

description: 'Формат номера исполнительного производства'

event:

must\_be\_in: ["Жалоба", "Заявление", "Отзыв"]

description: 'Разрешённые типы мероприятий'

adresat:

must\_be\_in: ["ФССП", "Сбербанк", "Почта Банк"]

description: 'Допустимые адресаты (обновлять по мере необходимости)'

**Используется для:**

* Валидации полей на этапе парсинга (validator.py).
* Контроля форматов даты, номера ИП и других ключевых реквизитов.
* Проверки на соответствие только разрешённым значениям (мероприятия, адресаты).

**Любые ошибки в правилах (битая регулярка, несуществующее поле) логируются как critical в error\_log.json, обработка таких документов невозможна.**

## **5.7. Связь справочников и скриптов**

* **main.py**
  + Инициирует последовательную загрузку и проверку всех справочников при старте процесса.
  + Если обнаружена ошибка/битый справочник/критическая валидация — немедленно завершает выполнение, пишет ошибку в error\_log.json.
  + После успешной проверки — запускает дальнейшие этапы обработки.
* **config.py**
  + Содержит функции load\_csv(), load\_yaml(), возвращает структуры справочников в память.
  + Отвечает за централизованный доступ к данным справочников для других модулей (singleton-подход или через DI).
* **validator.py**
  + Проверяет корректность структуры каждого справочника:
    - наличие обязательных полей,
    - отсутствие пустых строк/дубликатов,
    - соответствие регулярным выражениям (из validators.yaml).
  + Генерирует подробные сообщения обо всех найденных нарушениях и пишет их в error\_log.json.
* **state\_manager.py**
  + Отвечает за ведение журналов:
    - успешной загрузки справочников,
    - резервных копий при изменении справочников,
    - всех статусов справочников (загружен, битый, обновлён, резервная копия создана).
  + Позволяет восстановить/откатить справочник до последнего валидного состояния.
* **cli.py** (если реализован)
  + Предоставляет интерфейс ручной проверки и диагностики справочников:
    - запуск python cli.py validate-configs — валидирует все справочники без запуска обработки архива;
    - даёт краткий отчёт пользователю о статусе и ошибках, не затрагивая бизнес-логику.
* **Логика вызова и взаимодействия:**
  + **main.py** → вызывает **config.py** → (все справочники) → передаёт в **validator.py** → валидирует → **state\_manager.py** ведёт логи.
  + При любой ошибке или невалидности цепочка разрывается, ошибка уходит в журнал и процесс аварийно завершает работу.

# ****6. 📁 Итерация по архиву****

## **6.1. Алгоритм обхода кредиторов/должников**

**1. Получить финальный список кредиторов и должников:**

* Использовать данные из creditors\_to\_process.csv (статус “к обработке”).
* Исключить уже полностью обработанных (по process\_log.json).

**2. Для каждого кредитора:**

* Открыть сетевую/локальную папку (путь из поля link).
* Проверить доступ и целостность директории.

**3. Для каждого должника (подпапка кредитора):**

* Открыть папку должника.
* Проверить, что папка не пуста и доступна.
* Если нет папки — зафиксировать ошибку (“нет папки/нет доступа”) в error\_log.json.

**4. Рекурсивно обойти все вложенные папки должника:**

* Использовать модуль filewalker.py.
* На каждом уровне:
  + Собрать список всех файлов (с абсолютным путём).
  + Для каждой подпапки — повторить обход до самого “дна”.

**5. Для каждого найденного файла:**

* Определить расширение.
* Если файл не из списка разрешённых (formats.csv) —  
  — Зафиксировать в not\_processed.json с причиной (“неразрешённый формат”).  
  — Продолжить к следующему файлу.
* Если файл повреждён/не открывается —  
  — Зафиксировать ошибку открытия (тип ошибки, путь, имя файла, время) в error\_log.json.

**6. На выходе по каждому должнику:**

* Получить полный список файлов, допущенных к дальнейшей обработке.
* Сформировать отдельный журнал по “необработанным” и “ошибочным” файлам.

## **6.2. Валидация структуры и фиксация ошибок**

* **Проверки на каждом этапе обхода:**
  + Есть ли папка кредитора?
  + Есть ли у кредитора хотя бы один должник?
  + Есть ли у должника вложенные папки и/или файлы?
  + Все ли файлы доступны для чтения?
* **Если структура нарушена:**
  + Фиксация в error\_log.json с указанием уровня ошибки (кредитор, должник, конкретный файл).
  + Продолжение обработки остальных доступных элементов (skip only fail, never full stop).
* **Отчёт о структуре:**
  + В конце обхода — статистика по количеству:
    - Обработанных кредиторов/должников
    - Пустых папок
    - Ошибок доступа
    - Пропущенных файлов/директорий

## **6.3. Модули, отвечающие за итерацию и контроль:**

* **filewalker.py** — реализует рекурсивный обход, возвращает списки всех файлов и вложенных папок.
* **state\_manager.py** — фиксирует ошибки, неразобранные и пропущенные элементы.
* **validator.py** — контролирует корректность структуры папок и доступность файлов.

### ****Результат этапа:****

* Собраны и зафиксированы все файлы к обработке, ошибки структурирования, пропуски.
* Вся “белая зона” (обрабатываемые файлы) и “серая зона” (ошибки, недоступы, неразрешённые форматы) **жёстко зафиксированы для последующего аудита**.
* Можно переходить к анализу содержимого документов.

# ****7. 📝 Парсинг и анализ документов****

**7.1. Фильтрация файлов**

### 7.1.1. Вход: ожидания к формату входных данных

**Назначение:**  
На этом этапе система получает на вход список файлов, подготовленный после обхода архива и первичной фильтрации на уровне папок и кредиторов.  
Файлы могут быть в разных вложенных папках, иметь одинаковые имена, содержать разные форматы (PDF, DOCX, DOC, XLSX, JPG/PNG для сканов документов).

#### **Ожидания к формату входных данных:**

* **Источник данных:**
  + Генерируется модулем filewalker.py на предыдущем этапе.
  + Передаётся как список абсолютных путей к каждому найденному файлу:
  + files = [
  + "/data/Сбербанк/Иванов\_АА/IN/doc001.pdf",
  + "/data/Сбербанк/Иванов\_АА/IN/photo\_01.jpg",
  + "/data/ВТБ/Петров\_ПП/OUT/отчёт.docx",
  + # и т.д.
  + ]
* **Тип данных:**
  + Список строк (Python list of str), где каждая строка — это абсолютный путь к файлу.
* **Требования к каждому файлу:**
  + Файл существует физически (не битая ссылка).
  + Файл доступен для чтения (право доступа пользователя).
  + Размер файла больше нуля (не пустой файл).
  + Имя файла (basename) и путь к нему уникальны в контексте одной сессии обработки (допустимы одинаковые имена в разных папках).
* **Дополнительные ожидания:**
  + Файл соответствует одному из расширений из справочника /configs/formats.csv (например, pdf, docx, doc, xlsx, jpg, png).
  + Для каждого файла сохраняется “родительский” путь (кредитор, должник), чтобы в дальнейшем привязывать результаты к нужной сущности.
* **Пример структуры входных данных для парсера:**
* [
* {
* "filepath": "/data/Сбербанк/Иванов\_АА/IN/doc001.pdf",
* "creditor": "Сбербанк",
* "debtor": "Иванов\_АА",
* "parent\_folder": "IN"
* },
* ...
* ]

Такой формат удобен для логирования и дальнейшей работы с результатами.

#### **Резюме для документации:**

**На входе в этап фильтрации** всегда должен быть чётко определённый список файлов (пути + метаданные), который:

* Был получен автоматически на предыдущих шагах;
* Гарантированно физически существует;
* Может быть обработан в рамках поддерживаемых форматов;
* Связан с контекстом “кредитор-должник”.

### 7.1.2. Логика проверки расширения (formats.csv)

**Назначение:**  
На этом этапе из входного списка файлов отбираются только те, которые соответствуют разрешённым форматам (из /configs/formats.csv). Остальные автоматически помечаются как не подлежащие обработке и логируются.

#### **Пошаговый алгоритм:**

1. **Загрузка справочника форматов:**  
   Используется модуль config.py для чтения /configs/formats.csv.  
   Получается список разрешённых расширений (например, ["pdf", "docx", "doc", "xlsx", "jpg", "png"]).
2. # config.py
3. import csv
4. def load\_formats(config\_path):
5. formats = []
6. with open(config\_path, encoding="utf-8") as f:
7. reader = csv.DictReader(f)
8. for row in reader:
9. formats.append(row["extension"].lower())
10. return formats
11. **Проверка расширения каждого файла:**
    * Для каждого файла извлекается расширение (без точки, в нижнем регистре).
    * Сравнивается с разрешённым списком.
    * Если расширение не разрешено — файл сразу исключается из дальнейшей обработки.
12. # parser.py (пример)
13. import os
14. allowed\_exts = load\_formats("/configs/formats.csv")
15. for file in files:
16. ext = os.path.splitext(file["filepath"])[1][1:].lower()
17. if ext not in allowed\_exts:
18. # Логируем причину и пропускаем
19. state\_manager.log\_not\_processed(
20. file["filepath"], reason="неразрешённый формат", ext=ext
21. )
22. continue
23. # Далее — только для разрешённых файлов
24. **Логирование “неразрешённых” файлов:**  
    Все такие файлы фиксируются в /logs/not\_processed.json с указанием причины:
25. {
26. "datetime": "2024-06-21T10:23:00",
27. "file": "/data/Сбербанк/Иванов\_АА/IN/photo\_01.gif",
28. "reason": "неразрешённый формат",
29. "ext": "gif",
30. "creditor": "Сбербанк",
31. "debtor": "Иванов\_АА"
32. }

#### **Резюме для документации:**

* Все “чужие” форматы моментально исключаются до дальнейшей траты ресурсов.
* Вся история таких файлов хранится для последующего аудита (можно легко объяснить, почему что-то не попало в итоговую выгрузку).
* Этот этап — обязательная “санитарная проверка” входного архива.

### 7.1.3. Проверка доступности, пустых и битых файлов

**Назначение:**  
Отсеять файлы, которые физически недоступны, пусты (0 байт) или не читаются из-за ошибок файловой системы, битых секторов, блокировок или повреждений.

#### **Пошаговый алгоритм:**

1. **Проверка наличия файла на диске:**
   * Используется стандартная функция проверки пути (например, os.path.exists).
   * Если файл не найден — фиксируется ошибка “нет на диске”.
2. **Проверка размера файла:**
   * Если размер 0 байт — считается пустым, логируется как “пустой файл”.
3. **Проверка на доступность для чтения:**
   * Пробуем открыть файл на чтение (минимально — хотя бы 1 байт).
   * Ловим исключения:
     + PermissionError — нет доступа
     + OSError, IOError — другие ошибки (битый, занят процессом и т.д.)
   * Если файл не открывается — причина фиксируется.

**Пример кода:**

# parser.py

import os

def check\_file\_access(filepath):

if not os.path.exists(filepath):

return False, "файл не найден"

if os.path.getsize(filepath) == 0:

return False, "пустой файл"

try:

with open(filepath, "rb") as f:

f.read(1)

return True, ""

except PermissionError:

return False, "нет прав на чтение"

except Exception as e:

return False, f"ошибка чтения: {str(e)}"

for file in allowed\_files: # Только после фильтрации по расширению!

ok, msg = check\_file\_access(file["filepath"])

if not ok:

state\_manager.log\_error(

file["filepath"], error\_type="доступ/битый/пустой", msg=msg

)

continue

# Далее — только для реально читаемых файлов

1. **Логирование проблемных файлов:**
   * Все файлы, которые не прошли доступность или читаемость, фиксируются в /logs/error\_log.json:

{

"datetime": "2024-06-21T10:28:00",

"file": "/data/Сбербанк/Иванов\_АА/IN/contract.pdf",

"error\_type": "пустой файл",

"msg": "пустой файл",

"creditor": "Сбербанк",

"debtor": "Иванов\_АА"

}

#### **Резюме для документации:**

* На этом этапе отбрасываются все “пустышки” и недоступные файлы — они не попадут в дальнейший парсинг.
* Все причины пропуска детально фиксируются и доступны для аудита и последующей доработки (например, если причина — нет прав на папку).
* Экономится время и ресурсы: к обработке идут только реально существующие и читаемые файлы.

### 7.1.4. Уникализация и фиксация имени/пути файла

**Цель:**  
Обеспечить однозначную идентификацию каждого файла в процессе — независимо от совпадений имён, местоположения в структуре или количества исполнительных производств внутри.

#### **1. Фиксация контекста файла на этапе отбора**

* Для каждого найденного файла сохраняется:
  + session\_id — уникальный идентификатор текущей обработки (например, session\_20240621\_1030)
  + creditor — наименование кредитора (из структуры или справочника)
  + debtor\_folder — имя подпапки-должника (на этом этапе — “гипотеза”)
  + original\_filename
  + relative\_path — путь относительно /data/
  + absolute\_path — полный путь на диске

Все эти параметры формируют минимально уникальный “ключ” для каждого файла — вне зависимости от их содержимого.

#### **2. Не переименовывать файлы физически**

* Исходные файлы не переименовываются и не перемещаются.
* Для временных копий/выгрузок (если требуется) используется шаблон:
* {session\_id}\_{creditor}\_{debtor\_name}\_{number\_ip}\_{original\_filename}
* Вся идентификация на уровне логики и метаданных.

#### **3. После парсинга — дополняем “истинными” полями**

* После извлечения содержимого (парсинга):
  + debtor\_name — ФИО должника (из текста документа)
  + number\_ip — номер исполнительного производства (или несколько, если в документе более одного)
  + document\_date — дата документа
* Для каждого найденного ИП создаётся отдельная запись, сохраняется связка с исходным файлом и сессией.

#### **4. Контроль дубликатов и коллизий**

* Если совпадают relative\_path и original\_filename в одной папке — логируется предупреждение в error\_log.json.
* Для выгрузки: используется либо session\_id+number\_ip+date, либо md5-хэш файла (для полной уникальности).
* В процессе всё привязано к “сессии обработки”.

#### **5. Примеры структур**

**Перед парсингом:**

{

"session\_id": "session\_20240621\_1030",

"creditor": "Сбербанк",

"debtor\_folder": "Иванов\_АА",

"original\_filename": "contract.pdf",

"relative\_path": "Сбербанк/Иванов\_АА/IN/contract.pdf",

"absolute\_path": "/mnt/data/Сбербанк/Иванов\_АА/IN/contract.pdf"

}

**После парсинга:**

{

"session\_id": "session\_20240621\_1030",

"creditor": "Сбербанк",

"debtor\_folder": "Иванов\_АА",

"debtor\_name": "Иванов Алексей Андреевич",

"number\_ip": "123456/24/98765-ИП",

"document\_date": "2024-06-21",

"original\_filename": "contract.pdf",

"relative\_path": "Сбербанк/Иванов\_АА/IN/contract.pdf",

"status": "parsed"

}

#### **6. Фиксация в процесс-логах**

* На каждом этапе для каждого файла создаётся запись в process\_log.json с фиксацией статуса (“отобран к обработке”, “распарсен”, “отправлен в API” и т.д.).

#### **Резюме для документации:**

* **Уникальность обеспечивается:**  
  комбинацией session\_id + creditor + relative\_path + (number\_ip и document\_date после парсинга).
* **Оригинальное имя и путь** сохраняются всегда.
* **Переименование** производится только для временных или экспортируемых файлов, не влияет на исходные данные.
* **Легко реализуем аудит, восстановление и трассировку для любых сценариев.**

### 7.1.5. Алгоритм фильтрации: схема и пример кода

**Задача:**  
Обеспечить, чтобы на дальнейшие этапы попадали только корректные и допустимые к обработке файлы (PDF, DOC, DOCX и др. по справочнику).

#### **1. Логика фильтрации — схема**

1. **Для каждого файла в собранном списке:**
   * Проверить расширение файла по справочнику formats.csv
   * Проверить, доступен ли файл для чтения
   * Исключить “пустые” файлы (размер = 0 байт)
   * Исключить уже обработанные/дубли (по process\_log.json)
   * В случае несоответствия — записать причину в not\_processed.json/error\_log.json
2. **На выходе:**  
   Список файлов-”кандидатов” для парсинга, с уникальными путями и метаданными.

#### **2. Алгоритм фильтрации в виде блок-схемы**

[Старт] → [Собрать список файлов]

↓

[Для каждого файла:]

↓

[Файл существует?] — нет → [error\_log]

↓ да

[Допустимое расширение?] — нет → [not\_processed.json]

↓ да

[Доступен для чтения?] — нет → [error\_log]

↓ да

[Размер > 0?] — нет → [not\_processed.json]

↓ да

[Уже обработан?] — да → [skip]

↓ нет

[Добавить в список на обработку]

#### **3. Пример кода фильтрации**

import os

import pandas as pd

# Загрузка разрешённых расширений

formats = pd.read\_csv("configs/formats.csv")["extension"].tolist()

def filter\_files(file\_list, processed\_set):

filtered = []

for file\_meta in file\_list:

ext = file\_meta['original\_filename'].split('.')[-1].lower()

path = file\_meta['absolute\_path']

# Проверка 1: расширение

if ext not in formats:

log\_not\_processed(file\_meta, reason="Not allowed extension")

continue

# Проверка 2: существование и доступность

if not os.path.exists(path) or not os.access(path, os.R\_OK):

log\_error(file\_meta, error="File not accessible")

continue

# Проверка 3: размер

if os.path.getsize(path) == 0:

log\_not\_processed(file\_meta, reason="Empty file")

continue

# Проверка 4: обработан ранее

unique\_key = (file\_meta['relative\_path'], file\_meta['session\_id'])

if unique\_key in processed\_set:

continue # Пропустить дубли

# Всё хорошо — добавить в список кандидатов

filtered.append(file\_meta)

return filtered

#### **4. Что и где логируется**

* not\_processed.json — все недопустимые расширения, пустые файлы и пр.
* error\_log.json — физические ошибки доступа, отсутствие файлов и пр.

#### **5. Резюме**

* На обработку попадает только тот файл, который прошёл все фильтры.
* Все причины исключения (и технические, и бизнес-правила) строго фиксируются.
* Фильтрация происходит ДО любого анализа содержимого.

**7.2. Извлечение текста из файлов**

Отлично, вот чётко структурированный и практичный раздел по алгоритму выбора метода извлечения текста из файлов:

### 7.2. Извлечение текста из файлов

#### 7.2.1. Алгоритм выбора метода извлечения

(для PDF, DOCX, DOC, XLSX, изображений)

#### **1. Общая схема (Data Extraction Pipeline):**

[Вход: file\_meta] → [Определение расширения] → [Выбор метода:]

├─ PDF → [PDF parser] + [OCR при необходимости]

├─ DOCX → [python-docx] (структурированный текст)

├─ DOC → [textract/libreoffice] → [plain text]

├─ XLSX → [pandas.read\_excel] (вытаскиваем только если ожидается текст)

├─ PNG/JPG (или PDF со сканом) → [OCR (pytesseract/pdf2image)]

└─ иные → [Логируем как неподдерживаемый, пропуск]

#### **2. Логика выбора метода (псевдокод):**

def extract\_text(file\_meta):

ext = file\_meta['original\_filename'].split('.')[-1].lower()

path = file\_meta['absolute\_path']

if ext == 'pdf':

# 1. Попытаться извлечь текст через PyPDF2/pdfminer

text = extract\_pdf\_text(path)

if not text or is\_scanned\_pdf(path):

# 2. Если PDF “скан” (мало/нет текста) — запустить OCR (pytesseract)

text = extract\_pdf\_ocr(path)

elif ext == 'docx':

text = extract\_docx\_text(path)

elif ext == 'doc':

text = extract\_doc\_text(path) # через textract/libreoffice

elif ext == 'xlsx':

text = extract\_xlsx\_text(path)

elif ext in ['jpg', 'jpeg', 'png', 'tiff']:

text = extract\_image\_ocr(path)

else:

log\_not\_processed(file\_meta, reason="Unsupported file type")

return None

return text

#### **3. Для каждого типа — какой инструмент и что на выходе:**

| **Тип файла** | **Основной модуль** | **Альтернативы/резервы** | **Комментарий** |
| --- | --- | --- | --- |
| PDF | PyPDF2, pdfminer.six | OCR (pytesseract, pdf2image) | Если нет “живого” текста, используем OCR |
| DOCX | python-docx | — | Извлекает структурированный текст |
| DOC | textract, libreoffice | — | Может понадобиться конвертация в TXT |
| XLSX | pandas.read\_excel | openpyxl | Только для текстовых таблиц |
| PNG/JPG | pytesseract + pillow | — | OCR для “сканов” |

#### **4. Фиксация метода и исхода в логах**

* Для каждого файла фиксируется, каким методом был получен текст (source: "pdf+ocr", "docx", "image-ocr", и т.д.).
* Если текст не удалось извлечь — фиксируем причину в error\_log.json с указанием типа ошибки.

**Пример записи:**

{

"datetime": "2024-06-21T12:32:00",

"file": "Сбербанк/Иванов\_АА/IN/contract.pdf",

"method": "pdf+ocr",

"status": "success"

}

или

{

"datetime": "2024-06-21T12:33:00",

"file": "ПочтаБанк/Петров\_ВВ/IN/image1.jpg",

"method": "image-ocr",

"status": "fail",

"error": "OCR: no text found"

}

#### **5. Общий алгоритм:**

1. Получить путь и расширение файла.
2. Выбрать подходящий метод на основании расширения.
3. Извлечь текст, если возможно.
4. Сохранить исход метода, результат и статус в логи.
5. Если текст не получен — перейти к следующему файлу.

**7.2.2. Унификация результатов (str, encoding, pre-clean)**

**Задача:**  
Все тексты, извлечённые из разных форматов (PDF, DOCX, OCR, XLSX и др.), должны быть приведены к единому виду:

* **Тип:** Python-строка (str)
* **Кодировка:** UTF-8
* **Очистка:** Удаление невидимых символов, лишних пробелов, некорректных переносов, "мусора" OCR и пр.

**Пошаговый алгоритм (универсальный):**

1. **Гарантия типа str:**
   * Все возвращаемые данные из обработчиков должны быть явно приведены к str.
   * Если результат типа bytes — декодировать:
   * if isinstance(text, bytes):
   * text = text.decode('utf-8', errors='replace')
2. **Нормализация кодировки:**
   * Везде внутри проекта поддерживается только UTF-8.
   * Если возникает ошибка при декодировании — всегда использовать 'replace' для безопасного чтения.
3. **Предочистка (pre-clean):**
   * Удалить невидимые/непечатные символы (\u200b, \xa0, \ufeff, и т.п.).
   * Привести все переносы строк к \n, убрать дублирующиеся пробелы и пустые строки.
   * Для OCR: фильтровать артефакты (например, “~”, “\*”, “|”, если встречаются в абзацах без смысла).
   * Очистить хвосты “Page N”, мусор от колонтитулов и прочее — если есть характерные паттерны.
4. import re
5. def pre\_clean(text):
6. text = text.replace('\xa0', ' ').replace('\u200b', '').replace('\ufeff', '')
7. text = re.sub(r'\s+', ' ', text) # Много пробелов -> 1 пробел
8. text = re.sub(r'\n+', '\n', text) # Много переводов строк -> 1 перевод
9. text = text.strip()
10. return text
11. **Итог:**
    * Любой этап парсинга и извлечения текста (PDF, DOCX, OCR) завершает работу этим фильтром, прежде чем текст пойдёт в логику анализа/AI.
    * Если возникает ошибка обработки — она фиксируется в error\_log.json с текстом-исходником (для аудита).
    * Мини-тест: любой текст, переданный на дальнейшую обработку, всегда должен быть типа str, utf-8, без “мусора” и пустых абзацев.

**Резюме для документации:**

* Все тексты всегда приводятся к единому виду ещё ДО этапа поиска полей и отправки в AI.
* Унификация — ключевой шаг для кросс-форматной стабильности, поиска шаблонов и предотвращения багов с кодировками.

### 7.2.3. Пример кода для каждого формата

#### **PDF (text layer + OCR на сканах)**

from PyPDF2 import PdfReader

from pdf2image import convert\_from\_path

import pytesseract

from PIL import Image

def extract\_text\_pdf(file\_path):

# 1. Сначала пробуем извлечь текст обычным способом

reader = PdfReader(file\_path)

text = ''.join(page.extract\_text() or '' for page in reader.pages)

if text.strip():

return text # Если что-то есть — возвращаем

# 2. Если пусто — пробуем OCR по картинкам страниц

images = convert\_from\_path(file\_path)

ocr\_text = ''

for img in images:

ocr\_text += pytesseract.image\_to\_string(img, lang='rus+eng') + '\n'

return ocr\_text

#### **DOCX**

from docx import Document

def extract\_text\_docx(file\_path):

doc = Document(file\_path)

return '\n'.join([p.text for p in doc.paragraphs])

#### **DOC (старый Word) и универсально (textract)**

import textract

def extract\_text\_doc(file\_path):

text = textract.process(file\_path, encoding='utf-8')

return text.decode('utf-8', errors='replace')

#### **XLSX / CSV**

import pandas as pd

def extract\_text\_xlsx(file\_path):

xl = pd.ExcelFile(file\_path)

content = []

for sheet in xl.sheet\_names:

df = xl.parse(sheet)

content.append(df.to\_string(index=False, header=True))

return '\n\n'.join(content)

def extract\_text\_csv(file\_path):

df = pd.read\_csv(file\_path)

return df.to\_string(index=False, header=True)

#### **Изображения (JPG, PNG, TIFF, и пр.)**

from PIL import Image

import pytesseract

def extract\_text\_image(file\_path):

img = Image.open(file\_path)

return pytesseract.image\_to\_string(img, lang='rus+eng')

#### **Единая “обёртка” для авто-выбора:**

import os

def extract\_text\_auto(file\_path):

ext = os.path.splitext(file\_path)[-1].lower()

if ext in ('.pdf',):

return extract\_text\_pdf(file\_path)

elif ext in ('.docx',):

return extract\_text\_docx(file\_path)

elif ext in ('.doc',):

return extract\_text\_doc(file\_path)

elif ext in ('.xlsx',):

return extract\_text\_xlsx(file\_path)

elif ext in ('.csv',):

return extract\_text\_csv(file\_path)

elif ext in ('.jpg', '.jpeg', '.png', '.tif', '.tiff'):

return extract\_text\_image(file\_path)

else:

raise Exception(f"Unsupported file type: {ext}")

**NB:**

* После извлечения всегда прогонять результат через pre\_clean (см. 7.2.2)!
* Все ошибки — в error\_log.json, оригинальный файл и тип ошибки указывать обязательно.

### 7.2.4. Ошибки извлечения — логика и фиксация

**Цель:**  
Любая ошибка на этапе извлечения текста фиксируется, чтобы не было “потерянных” файлов и чтобы любой кейс мог быть проанализирован вручную.

#### **Ключевые правила:**

1. **Все исключения/ошибки при попытке извлечения — через try/except.**
2. **Логировать абсолютно всё:** путь, формат, тип ошибки, traceback (если есть), сессию и статус (“extract\_error”).
3. **Не падать на ошибке — продолжать обработку других файлов.**
4. **Оригинальный файл НЕ удалять и НЕ перемещать!**
5. **Добавлять флаг status="ошибка\_извлечения" в структуру итоговой записи.**

#### **Пример кода фиксации ошибки:**

import traceback

import datetime

import json

def log\_extraction\_error(file\_path, session\_id, error, error\_log\_path):

entry = {

"datetime": datetime.datetime.now().isoformat(),

"session\_id": session\_id,

"file": file\_path,

"stage": "extract\_text",

"status": "ошибка\_извлечения",

"error": str(error),

"traceback": traceback.format\_exc()

}

# Добавляем в error\_log.json

with open(error\_log\_path, "a", encoding="utf-8") as f:

f.write(json.dumps(entry, ensure\_ascii=False) + "\n")

#### **Встраивание в обработку:**

try:

text = extract\_text\_auto(file\_path)

except Exception as e:

log\_extraction\_error(

file\_path=file\_path,

session\_id=session\_id,

error=e,

error\_log\_path="logs/error\_log.json"

)

text = ""

# Также можно добавить запись в not\_processed.json

#### **Чек-лист по обработке ошибок:**

* Ошибка зафиксирована в error\_log.json (сессия, путь, причина, stacktrace).
* В итоговой структуре — status="ошибка\_извлечения".
* Исходный файл НЕ трогается.
* Логирование не прерывает цикл обработки.
* Для анализа ошибок есть поиск по файлу, сессии, типу ошибки.

**Итого:**  
— Любая проблема при извлечении текста не “теряет” файл, а сразу видна оператору/аналитику.

**7.3. Нормализация имени файла, уникальность**

## 🧾 **7.3.1. Алгоритм создания уникального имени (ID): дата обработки, номер ИП, индекс в сессии**

### 🎯 ****Назначение:****

Обеспечить **однозначную идентификацию каждого документа**, даже если:

* В сессии обрабатываются десятки документов с одинаковым номером ИП;
* Повторная обработка запускается вручную;
* Требуется трассировка по логам, выгрузкам или повторным обращениям.

### ⚙️ ****Структура уникального имени файла:****

<дата обработки>\_<номер ИП>\_<индекс в сессии>.pdf

| **Компонент** | **Формат** | **Пример** | **Источник** |
| --- | --- | --- | --- |
| Дата | YYYYMMDD | 20250622 | datetime.now() |
| Номер ИП | 123456/24/98765-ИП | 123456-24-98765-ИП | Из документа, нормализован |
| Индекс в сессии | 001, 002, 003... | 001 | Порядковый номер обработки |

📌 **Важно:** номер ИП должен быть очищен от слэшей и заменён на дефисы для совместимости с файловыми системами.

### 🔄 ****Пример имени:****

20250622\_123456-24-98765-ИП\_002.pdf

### 🧠 ****Особенности и нюансы:****

* Имя используется при **сохранении файла**, **логировании**, **выгрузке в JSON/Excel**.
* Для **одного и того же документа**, обработанного повторно, может отличаться только индекс.
* Поддерживает **трассировку сессий и партиционирование выгрузок**.

### 💻 ****Реализация (Python):****

from datetime import datetime

import re

def normalize\_ip\_number(ip\_number: str) -> str:

"""Преобразует номер ИП в файловый-safe формат."""

return re.sub(r'[^\w\-]', '-', ip\_number.strip())

def generate\_unique\_filename(ip\_number: str, index: int, ext: str = ".pdf") -> str:

date\_str = datetime.now().strftime("%Y%m%d")

norm\_ip = normalize\_ip\_number(ip\_number)

return f"{date\_str}\_{norm\_ip}\_{index:03}{ext}"

✅ **Пример вызова:**

generate\_unique\_filename("123456/24/98765-ИП", 2)

# → 20250622\_123456-24-98765-ИП\_002.pdf

### 📎 ****Точки использования:****

* exporter.py — при сохранении итогового документа или JSON.
* state\_manager.py — при логировании.
* parser.py — при фиксации результата.

Отлично, ты прав — логика версий (\_v2, \_v3) **не соответствует реальной стратегии проекта**, где:

* Каждый документ уникален по **дате обработки**;
* Повторная обработка **не должна** записываться под тем же именем, а должна **создавать новую запись с другой датой**;
* Идентификатор должен быть основан на **дата + номер ИП + индекс** (в рамках одной сессии).

## 🧾 **7.3.2. Проверка на коллизии: акцент на дату, без версий**

### 🎯 ****Назначение:****

Гарантировать, что имя каждого сохраняемого файла **однозначно указывает на его дату обработки**, даже если:

* Один и тот же документ обрабатывается повторно;
* В архиве присутствует несколько копий одного ИП;
* Проводится пакетная обработка с одним номером ИП, но разными событиями или датами.

### ⚠️ ****Что считаем коллизией:****

Коллизией **не** считается совпадение номера ИП и индекса.  
Коллизия — это если **имя полностью совпадает**, включая дату (что означает сбой генерации имени).

📌 А при корректной архитектуре каждый запуск формирует:

<текущая дата>\_<нормализованный номер ИП>\_<индекс>.pdf

Что **исключает коллизии**, даже при повторной обработке одного и того же документа в разные дни.

### ✅ ****Требуемая стратегия:****

* **Не использовать** постфиксы вроде \_v2, \_v3.
* Всегда формировать новое имя на основе **текущей даты**, даже если файл обрабатывался ранее.
* Коллизия возможна только если в рамках одной сессии создаётся дважды файл с одинаковым ИП и индексом — тогда индекс инкрементируется.

### 💻 ****Пример кода: строгое формирование имени и предотвращение коллизий:****

import os

from datetime import datetime

import re

def normalize\_ip\_number(ip\_number: str) -> str:

"""Преобразует номер ИП в безопасный для имени файла вид"""

return re.sub(r'[^\w\-]', '-', ip\_number.strip())

def generate\_unique\_filename(ip\_number: str, session\_index: int, ext: str = ".pdf") -> str:

"""

Генерирует уникальное имя по дате, номеру ИП и индексу.

Коллизия исключена при корректной сессии.

"""

date\_str = datetime.now().strftime("%Y%m%d")

ip\_str = normalize\_ip\_number(ip\_number)

return f"{date\_str}\_{ip\_str}\_{session\_index:03}{ext}"

### 🧩 ****Контроль на этапе сохранения:****

Перед сохранением файла всё равно можно убедиться, что **такого имени не существует в целевой папке**, и при совпадении:

* Либо повысить session\_index;
* Либо зафиксировать логическую ошибку (если повторное имя — это ошибка сценария).

### 📎 ****Примеры имён:****

| **Имя файла** | **Дата** | **Номер ИП** | **Индекс** |
| --- | --- | --- | --- |
| 20250622\_123456-24-98765-ИП\_001.pdf | 22.06.2025 | 123456/24/98765-ИП | 001 |
| 20250622\_123456-24-98765-ИП\_002.pdf | 22.06.2025 | тот же | 002 |
| 20250623\_123456-24-98765-ИП\_001.pdf | 23.06.2025 | тот же | 001 — новая дата, новый файл |

## 🧾 **7.3.3. Унификация расширений и форматов**

### 🎯 ****Назначение:****

Привести все допустимые расширения входных файлов к **одному внутреннему стандарту** (например, .pdf) и обеспечить **предсказуемость поведения** парсера и логов.

### 📦 ****Входные данные:****

* Файлы могут иметь расширения: .pdf, .PDF, .Pdf, .docx, .DOCX, .doc, .jpeg, .jpg и пр.
* Требуется унифицировать:
  + регистр (PDF → pdf);
  + формат (doc, jpeg → предварительная конвертация или исключение);
  + структура путей и имён файлов до начала анализа.

### ⚠️ ****Типовые проблемы:****

| **Проблема** | **Последствия** |
| --- | --- |
| .PDF vs .pdf | могут быть восприняты как разные |
| .doc без поддержки | ошибка при чтении |
| .jpeg без OCR | пустой результат |
| file без расширения | невозможность определить тип, сбой |

### ✅ ****Решение — этап предварительной нормализации:****

1. **Приведение расширения к нижнему регистру**
2. **Фильтрация разрешённых форматов по formats.csv**
3. **Удаление или логирование неподдерживаемых**
4. **(Опционально) Автоконвертация doc → docx и jpeg → PDF/OCR**

### 💻 ****Пример реализации (Python):****

import os

def normalize\_extension(filename: str) -> str:

"""Приводит расширение к нижнему регистру."""

base, ext = os.path.splitext(filename)

return base + ext.lower()

def is\_extension\_allowed(filename: str, allowed\_exts: set[str]) -> bool:

"""Проверяет, входит ли расширение в список допустимых."""

ext = os.path.splitext(filename)[-1][1:].lower()

return ext in allowed\_exts

### 📎 ****Сценарий в рамках обработки:****

1. filewalker.py:
   * Приводит расширения всех найденных файлов к нижнему регистру.
   * Удаляет/логирует всё, что не входит в formats.csv.
2. not\_processed.json:
   * Сохраняет путь и причину исключения, например:
   * {
   * "datetime": "2025-06-22T14:32:00",
   * "file": "/data/123/письмо.jpeg",
   * "reason": "unsupported\_extension"
   * }
3. config.py:
   * Загружает справочник допустимых расширений из formats.csv.

### 📋 ****Контрольные точки:****

* ❗ Перед парсингом ни один “сырой” файл не должен попасть в parser.py без унифицированного расширения.
* 📤 Выгрузка/лог обязательно содержит нормализованное имя файла.

### 🧾 Пример до / после нормализации:

| **Исходный файл** | **После нормализации** |
| --- | --- |
| Документ.PDF | Документ.pdf |
| memo.DOC | memo.doc → memo.docx (если есть автоконвертация) |
| 1234\_жалоба.JPEG | не обработан (в лог) |

### 🧩 Модули:

* filewalker.py — нормализует расширение и фильтрует
* config.py — загружает список допустимых
* state\_manager.py — логирует “не обрабатываемые” файлы
* parser.py — работает только с отфильтрованными, унифицированными

## 🧾 **7.3.4. Переименование обработанного файла и/или запись маппинга в журнал**

### 🎯 ****Назначение:****

После успешной обработки документа система должна:

* либо переименовать файл в соответствии с внутренним ID (дата\_номерИП\_индекс);
* либо сохранить маппинг: **“оригинальное имя → нормализованное имя”** в журнал для последующего контроля и аудита.

Это исключает повторную обработку и позволяет отслеживать, какие файлы были привязаны к какому результату.

### 📌 ****Варианты поведения:****

| **Поведение** | **Когда применяется** |
| --- | --- |
| Переименование файла | Локальные файлы, над которыми есть полный контроль |
| Запись в журнал name\_map.json | Если переименование невозможно (сетевые хранилища, ограниченные права) |

### 💡 ****Формат маппинга (пример):****

{

"original\_file": "C:/data/sber/ip123/жалоба.PDF",

"normalized\_file": "20250622\_123456-24-98765-ИП\_001.pdf",

"ip\_number": "123456/24/98765-ИП",

"index": 1,

"datetime": "2025-06-22T14:41:00"

}

### 💻 ****Пример кода:****

import json

from datetime import datetime

import os

def log\_name\_mapping(original\_path, normalized\_name, ip\_number, index, log\_path='logs/name\_map.json'):

entry = {

"original\_file": original\_path,

"normalized\_file": normalized\_name,

"ip\_number": ip\_number,

"index": index,

"datetime": datetime.now().isoformat()

}

with open(log\_path, 'a', encoding='utf-8') as f:

f.write(json.dumps(entry, ensure\_ascii=False) + '\n')

### 🧩 ****Вариант с переименованием:****

import os

import shutil

def rename\_processed\_file(original\_path, normalized\_name, destination\_folder):

new\_path = os.path.join(destination\_folder, normalized\_name)

shutil.move(original\_path, new\_path)

return new\_path

### 📎 ****Что логируется:****

* Оригинальный путь к файлу
* Имя после нормализации
* Дата и время переименования
* Номер ИП и индекс в сессии

### 📂 ****Файл хранения маппинга:****

* logs/name\_map.json — журнал соответствия оригинальных и финальных имён
* Используется для: аудита, поиска источника ошибки, повторного анализа

### 📋 ****Контрольные условия:****

* ✅ Переименование допустимо только при наличии прав записи в директорию
* ✅ Если переименование невозможно — обязательна запись в журнал
* ❌ Никогда не переименовывать, если файл загружен с сетевого ресурса с read-only правами

## 🧾 **7.3.5. Пример кода/функции по генерации уникального имени файла**

(примыкает к 7.3.1–7.3.4, объединяет всё в цепочку)

### 🎯 ****Назначение:****

Функция должна выполнять **полную генерацию уникального имени файла**, опираясь на:

* дату запуска обработки,
* номер ИП,
* индекс внутри сессии,
* логирование соответствия оригинального пути и нормализованного имени.

Также — проверять корректность входных параметров и логировать результат в logs/name\_map.json.

### ⚙️ ****Обязательные действия внутри функции:****

1. Очистить номер ИП от недопустимых символов.
2. Преобразовать дату в формат YYYYMMDD.
3. Собрать имя по шаблону <дата>\_<нормализованный\_номер\_ИП>\_<индекс>.pdf.
4. Вернуть итоговое имя и при необходимости — записать маппинг в журнал.

### 💻 ****Полный код функции — без сокращений:****

import os

import re

import json

from datetime import datetime

def normalize\_ip\_number(ip\_number: str) -> str:

"""

Нормализует номер исполнительного производства для использования в имени файла:

заменяет недопустимые символы (/, пробелы, табы и т.д.) на дефисы.

Пример: "123456/24/98765-ИП" → "123456-24-98765-ИП"

"""

cleaned = ip\_number.strip()

cleaned = re.sub(r'[^\w\-]', '-', cleaned) # заменяем всё, кроме букв/цифр/дефисов

return cleaned

def generate\_unique\_filename(ip\_number: str, index: int, extension: str = ".pdf") -> str:

"""

Генерирует уникальное имя файла по шаблону:

<дата обработки>\_<номер ИП нормализованный>\_<индекс>.pdf

"""

if not ip\_number:

raise ValueError("Номер ИП не может быть пустым")

if index < 1:

raise ValueError("Индекс файла должен быть >= 1")

date\_str = datetime.now().strftime("%Y%m%d")

norm\_ip = normalize\_ip\_number(ip\_number)

filename = f"{date\_str}\_{norm\_ip}\_{index:03}{extension}"

return filename

def log\_name\_mapping(original\_path: str, final\_filename: str, ip\_number: str, index: int, log\_dir: str = "logs"):

"""

Записывает соответствие между оригинальным путём файла и сгенерированным именем в журнал name\_map.json.

"""

os.makedirs(log\_dir, exist\_ok=True)

log\_path = os.path.join(log\_dir, "name\_map.json")

entry = {

"datetime": datetime.now().isoformat(),

"original\_file": original\_path,

"normalized\_file": final\_filename,

"ip\_number": ip\_number,

"index": index

}

with open(log\_path, 'a', encoding='utf-8') as f:

f.write(json.dumps(entry, ensure\_ascii=False) + '\n')

def process\_and\_name\_file(original\_path: str, ip\_number: str, index: int, output\_dir: str = "exports") -> str:

"""

Объединяющая функция:

1. Генерирует уникальное имя файла.

2. Записывает маппинг.

3. Возвращает итоговое имя (без физического переименования файла).

"""

final\_filename = generate\_unique\_filename(ip\_number, index)

log\_name\_mapping(original\_path, final\_filename, ip\_number, index)

return final\_filename

### 📂 ****Пример использования в цепочке обработки:****

# Где-то в parser.py или exporter.py

ip\_number = "123456/24/98765-ИП"

index = 1

original\_path = "/data/Сбербанк/ip001/жалоба.PDF"

new\_name = process\_and\_name\_file(original\_path, ip\_number, index)

# → new\_name = "20250622\_123456-24-98765-ИП\_001.pdf"

### 🧩 ****Связь с архитектурой:****

| **Модуль** | **Функция** |
| --- | --- |
| parser.py | вызывает process\_and\_name\_file() после извлечения ИП |
| exporter.py | использует имя при сохранении выгрузки |
| state\_manager.py | логирует через name\_map.json |
| logs/ | получает структуру трассировки |

**7.4. Извлечение полей, шаблоны и AI**

## 🧾 **7.4.1. Алгоритм поиска по регуляркам и справочникам (**validators.yaml**,** events.csv**,** adresat.csv**)**

### 🎯 ****Назначение:****

Обеспечить **надежное извлечение ключевых полей** из текста документа на основе:

* регулярных выражений (для дат, номеров ИП и др.);
* справочников (типов мероприятий, адресатов, фамилий ответственных).

### 📦 ****Входные данные:****

* **Исходный текст**, полученный из PDF/DOCX после OCR или текстовой выборки.
* **Файлы-справочники:**
  + validators.yaml — шаблоны регулярных выражений (дата, номер ИП и пр.).
  + events.csv — список допустимых названий мероприятий.
  + adresat.csv — список адресатов и кодов.
  + responsible.csv — ФИО и должности (если используется поиск ответственного).

### 📌 ****Ключевые цели извлечения:****

| **Поле** | **Метод извлечения** | **Обязательность** |
| --- | --- | --- |
| date | регулярка из validators.yaml | ✅ |
| number\_ip | регулярка из validators.yaml | ✅ |
| adresat | поиск по adresat.csv | ✅ |
| type\_event | ключевые слова из events.csv | ✅ |
| responsible | сопоставление ФИО из responsible.csv | ⛔ (опционально) |

### 📚 ****Пример содержимого**** validators.yaml****:****

date:

regex: '^(\d{2})\.(\d{2})\.(\d{4})$'

description: 'Дата в формате ДД.ММ.ГГГГ'

number\_ip:

regex: '^\d{6}/\d{2}/\d{5}-ИП$'

description: 'Формат номера ИП'

adresat:

must\_be\_in: ['ФССП', 'Сбербанк', 'Почта Банк']

type\_event:

keywords: ['Жалоба', 'Заявление', 'Отзыв']

### 💻 ****Пример функции поиска по регулярке:****

import re

def extract\_by\_regex(text: str, pattern: str) -> str | None:

"""

Ищет первое совпадение по регулярному выражению.

Возвращает найденное значение или None.

"""

match = re.search(pattern, text)

return match.group(0) if match else None

### 💻 ****Пример поиска по справочнику (адресат, мероприятие):****

def find\_by\_dictionary(text: str, dictionary\_values: list[str]) -> str | None:

"""

Возвращает первое найденное совпадение из справочника.

"""

for word in dictionary\_values:

if word.lower() in text.lower():

return word

return None

### 🔄 ****Полный вызов логики:****

def extract\_fields(text: str, validators: dict, events: list[str], adresaty: list[str]) -> dict:

fields = {}

fields["date"] = extract\_by\_regex(text, validators["date"]["regex"])

fields["number\_ip"] = extract\_by\_regex(text, validators["number\_ip"]["regex"])

fields["adresat"] = find\_by\_dictionary(text, adresaty)

fields["type\_event"] = find\_by\_dictionary(text, events)

return fields

### 📋 ****Формат извлечённого результата:****

{

"date": "15.06.2024",

"number\_ip": "123456/24/98765-ИП",

"adresat": "ФССП",

"type\_event": "Жалоба"

}

### 🧩 ****Используется в модулях:****

| **Модуль** | **Назначение** |
| --- | --- |
| parser.py | Запускает полный процесс извлечения |
| validator.py | Проверяет наличие ключевых полей после извлечения |
| ai\_client.py | Обрабатывает только после базового заполнения |

## 🧾 **7.4.2. Обработка шаблонов (поиск дат, номера ИП, адресата и т.д.)**

### 🎯 ****Назначение:****

Реализовать пошаговую и надёжную **обработку шаблонов полей**, на основе которых строится логика анализа и загрузки в 1С:

* дата;
* номер ИП;
* адресат;
* тип мероприятия.

Каждое поле должно быть извлечено независимо, проверено и нормализовано для включения в итоговую структуру выгрузки.

### 📌 ****Алгоритм работы по каждому полю:****

#### 🔹 1. Дата документа ("date")

* **Источник**: текст заголовка, шапка, строка с датой документа.
* **Регулярка**: validators.yaml → date.regex  
  Пример: ^(\d{2})\.(\d{2})\.(\d{4})$
* **Порядок действий:**
  1. Ищем первое совпадение по регулярке.
  2. Проверяем валидность (не 00.00.0000, не будущее).
  3. Конвертируем в нужный формат (если требуется → ISO, DD.MM.YYYY).

import re

def extract\_date(text: str, date\_pattern: str) -> str | None:

match = re.search(date\_pattern, text)

if match:

return match.group(0)

return None

#### 🔹 2. Номер исполнительного производства ("number\_ip")

* **Источник**: тело документа, строки с формулировкой "исполнительное производство".
* **Регулярка**: validators.yaml → number\_ip.regex  
  Пример: ^\d{6}/\d{2}/\d{5}-ИП$
* **Порядок действий:**
  1. Поиск по регулярке.
  2. Проверка корректности формата.
  3. Очистка от пробелов, лишних символов (если встречаются опечатки — сохраняем как есть, но логируем отклонение).

def extract\_number\_ip(text: str, ip\_pattern: str) -> str | None:

match = re.search(ip\_pattern, text)

return match.group(0) if match else None

#### 🔹 3. Адресат ("adresat")

* **Источник**: начало документа, обращение, служебная часть ("в адрес", "в ФССП", "в банк").
* **Словарь**: adresat.csv
* **Порядок действий:**
  1. Преобразование текста в нижний регистр.
  2. Поиск вхождения любого значения из справочника.
  3. Если найдено несколько — выбрать первое.
  4. При отсутствии — вернуть None, но сохранить в лог.

def extract\_adresat(text: str, adresat\_list: list[str]) -> str | None:

for a in adresat\_list:

if a.lower() in text.lower():

return a

return None

#### 🔹 4. Тип мероприятия ("type\_event")

* **Источник**: заголовок или ключевые слова в теле документа.
* **Словарь**: events.csv
* **Порядок действий:**
  1. Ищем совпадение по ключевым словам (например, "заявление", "жалоба").
  2. Совпадение должно быть точным (по слову или устойчивой фразе).
  3. Если найдено — возвращаем нормализованное значение.

def extract\_event(text: str, event\_keywords: list[str]) -> str | None:

for word in event\_keywords:

if word.lower() in text.lower():

return word

return None

### 🧩 ****Объединённый процесс (внутри parser.py):****

def extract\_document\_fields(text: str, validators: dict, events: list[str], adresaty: list[str]) -> dict:

return {

"date": extract\_date(text, validators["date"]["regex"]),

"number\_ip": extract\_number\_ip(text, validators["number\_ip"]["regex"]),

"adresat": extract\_adresat(text, adresaty),

"type\_event": extract\_event(text, events)

}

### 📋 ****Особенности:****

* Если не найдено поле number\_ip или date — документ считается **неполным**.
* Все результаты должны быть **логированы** в process\_log.json или error\_log.json в зависимости от результата.
* Ошибки извлечения — это **не сбой**, а метка "status": "неполные данные".

Вот **обновлённая и полностью выверенная версия раздела** 7.4.3, соответствующая твоему ТЗ, логике 1С и всей цепочке обработки:

## 🧾 **7.4.3. Выделение смысловых фрагментов (AI / шаблоны)**

### 🎯 ****Назначение:****

Использовать искусственный интеллект для **анализа уже извлечённого текста документа** и формирования **структурированного JSON-ответа**, который заполняет поля, соответствующие формату загрузки в 1С.

ИИ должен **одним запросом** вернуть заполненные значения для следующих полей:

### 📌 ****Целевые поля на выходе:****

| **Поле** | **Обязательность** | **Описание** |
| --- | --- | --- |
| Серия ИП | ✅ | Номер исполнительного производства (формат: 123456/24/98765-ИП) |
| Дата | ✅ | Дата документа (ДД.ММ.ГГГГ) |
| Мероприятие | ✅ | Название документа |
| № ИД | ✅ | Дублирует Серия ИП, используется в выгрузке |
| Адресат мероприятия | ✅ | Кому направлен документ (ФССП, банк и т.д.) |
| Адрес мероприятия | ⛔ | Адрес из текста (если есть — после слова "адрес", "по адресу", и т.д.) |
| Тип мероприятия | ✅ | Тип действия (Жалоба, Уведомление, Заявление) |
| Цель мероприятия | ✅ | Формулировка после слова "прошу", краткая и по сути |
| Комментарий о типе | ⛔ | Пока не используется, можно пропускать |
| Комментарий | ⛔ | Краткое резюме сути документа (1–2 строки) |
| Ответственный | ⛔ | Только если найдено ФИО из справочника |

### 🔁 ****Поведение при отсутствии данных:****

* Поля, которые не удалось определить, возвращаются как null или не включаются.
* Ответственный заполняется **только** при совпадении с одним из 4 допустимых ФИО из справочника.

### 🧠 ****Пример целевого JSON от ИИ:****

{

"Серия ИП": "123456/24/98765-ИП",

"Дата": "14.06.2024",

"Мероприятие": "Жалоба на действия пристава",

"№ ИД": "123456/24/98765-ИП",

"Адресат мероприятия": "ФССП",

"Адрес мероприятия": "г. Вологда, ул. Мира, д. 10",

"Тип мероприятия": "Жалоба",

"Цель мероприятия": "Проверить действия пристава",

"Комментарий о типе": null,

"Комментарий": "Заявитель жалуется на незаконное удержание денежных средств",

"Ответственный": "Иванов И.И."

}

### 🔍 ****Формирование prompt для ИИ (внутри**** ai\_client.py****):****

def build\_structured\_prompt(document\_text: str) -> str:

return f"""

Ты анализируешь текст официального документа. На выходе ты должен вернуть JSON-объект, заполнив только те поля, которые найдены в тексте.

Структура ответа:

{{

"Серия ИП": "...",

"Дата": "...",

"Мероприятие": "...",

"№ ИД": "...",

"Адресат мероприятия": "...",

"Адрес мероприятия": "...",

"Тип мероприятия": "...",

"Цель мероприятия": "...",

"Комментарий о типе": null,

"Комментарий": "...",

"Ответственный": "..."

}}

Текст документа:

\"\"\"{document\_text}\"\"\"

Верни только JSON. Если данных нет — возвращай null или не включай поле.

"""

### 💻 ****Интеграция и вызов:****

| **Модуль** | **Назначение** |
| --- | --- |
| parser.py | Передаёт текст и вызывает AI |
| ai\_client.py | Генерирует prompt, отправляет, парсит ответ |
| validator.py | Проверяет соответствие JSON-ответа схеме |
| state\_manager.py | Логирует JSON-ответ и мета-данные обработки |

### ⚠️ ****Обработка ошибок:****

* Если модель вернула некорректный JSON → логируется в error\_log.json.
* Если ключевые поля (Серия ИП, Дата, Мероприятие) отсутствуют — документ получает статус "неполные данные".
* В случае ошибки ответа ИИ — fallback не применяется, фиксируется как ai\_status: error.

.

## 🧾 **7.5. Проверка полноты, дублей, статус**

### 🎯 ****Назначение:****

После анализа документа и формирования JSON-записи нужно провести финальную **валидацию**, определить:

* все ли обязательные поля заполнены;
* нет ли дубля в рамках сессии;
* какой статус присвоить (норма / дубликат / неполные данные).

## 📌 **7.5.1. Валидация на обязательные поля**

**Обязательные поля:**

* Серия ИП
* Дата
* Мероприятие
* Адресат мероприятия
* Тип мероприятия
* Цель мероприятия

Если **хотя бы одно поле отсутствует или null**, запись считается **неполной**.

REQUIRED\_FIELDS = [

"Серия ИП",

"Дата",

"Мероприятие",

"Адресат мероприятия",

"Тип мероприятия",

"Цель мероприятия"

]

def is\_complete(record: dict) -> bool:

return all(record.get(field) for field in REQUIRED\_FIELDS)

## 📌 **7.5.2. Проверка на дубли по номеру ИП, дате и имени**

Дубликат — это запись, где **одинаковы**:

* Серия ИП
* Дата
* Тип мероприятия

Проверка ведётся по JSON-объектам, уже обработанным в этой сессии (в оперативной структуре session\_records).

def is\_duplicate(record: dict, session\_records: list[dict]) -> bool:

for existing in session\_records:

if (

existing.get("Серия ИП") == record.get("Серия ИП") and

existing.get("Дата") == record.get("Дата") and

existing.get("Тип мероприятия") == record.get("Тип мероприятия")

):

return True

return False

## 📌 **7.5.3. Пометка "неполные данные", логирование в отдельный журнал**

| **Статус** | **Условие** | **Действие** |
| --- | --- | --- |
| "ok" | Все обязательные поля заполнены, не дубликат | Добавить в results.json |
| "duplicate" | Полный, но уже существует | Добавить в duplicates.json |
| "incomplete" | Нет хотя бы одного обязательного поля | Добавить в incomplete.json, логировать |

def assign\_status(record: dict, session\_records: list[dict]) -> str:

if not is\_complete(record):

return "incomplete"

elif is\_duplicate(record, session\_records):

return "duplicate"

return "ok"

## 💾 **7.5.4. Пример итогового кода с логированием:**

import json

import os

def save\_record(record: dict, status: str, output\_dir: str = "logs/validated"):

os.makedirs(output\_dir, exist\_ok=True)

path = os.path.join(output\_dir, f"{status}.json")

with open(path, 'a', encoding='utf-8') as f:

f.write(json.dumps(record, ensure\_ascii=False) + '\n')

### 📂 Пример результата в журнале incomplete.json:

{

"Серия ИП": "123456/24/98765-ИП",

"Дата": null,

"Мероприятие": "Жалоба",

"Тип мероприятия": "Жалоба",

"Цель мероприятия": "Проверить действия пристава",

"status": "incomplete"

}

### 🧩 Интеграция:

| **Модуль** | **Назначение** |
| --- | --- |
| validator.py | Запуск всех проверок и назначение статуса |
| state\_manager.py | Сохранение по статусам |
| results/ | Три отдельных файла: ok, incomplete, duplicate |

## 🧾 **7.6. Итоговая структура на выходе**

### 🎯 ****Назначение:****

Для каждого **валидного** (status = "ok") документа необходимо:

* сформировать финальную структуру-словарь (dict) для сохранения и передачи;
* сохранить её в виде JSON-файла;
* отправить результат на следующий этап обработки (например, загрузка в 1С или в Excel-отчёт).

## 📌 **7.6.1. Формирование одной записи-словаря (dict) для валидного файла**

Каждый успешно обработанный файл формирует **один dict-объект**, включающий:

* все извлечённые и нормализованные поля;
* дополнительные мета-данные (время обработки, путь к оригинальному файлу, уникальный ID).

def build\_final\_record(record\_data: dict, original\_path: str, normalized\_name: str) -> dict:

return {

"Серия ИП": record\_data.get("Серия ИП"),

"Дата": record\_data.get("Дата"),

"Мероприятие": record\_data.get("Мероприятие"),

"№ ИД": record\_data.get("Серия ИП"),

"Адресат мероприятия": record\_data.get("Адресат мероприятия"),

"Адрес мероприятия": record\_data.get("Адрес мероприятия"),

"Тип мероприятия": record\_data.get("Тип мероприятия"),

"Цель мероприятия": record\_data.get("Цель мероприятия"),

"Комментарий о типе": record\_data.get("Комментарий о типе"),

"Комментарий": record\_data.get("Комментарий"),

"Ответственный": record\_data.get("Ответственный"),

"Файл": normalized\_name,

"Оригинал": original\_path,

"Обработано": datetime.now().isoformat(),

"Status": "ok"

}

## 📌 **7.6.2. Структура записи (JSON)**

### ✅ Пример итоговой JSON-записи:

{

"Серия ИП": "123456/24/98765-ИП",

"Дата": "14.06.2024",

"Мероприятие": "Жалоба",

"№ ИД": "123456/24/98765-ИП",

"Адресат мероприятия": "ФССП",

"Адрес мероприятия": "г. Вологда, ул. Мира, д. 10",

"Тип мероприятия": "Жалоба",

"Цель мероприятия": "Проверить действия пристава",

"Комментарий о типе": null,

"Комментарий": "Жалоба на удержание средств",

"Ответственный": "Иванов И.И.",

"Файл": "20250622\_123456-24-98765-ИП\_001.pdf",

"Оригинал": "C:/data/sber/ip123/жалоба.PDF",

"Обработано": "2025-06-22T14:40:10",

"Status": "ok"

}

## 📌 **7.6.3. Передача результата на следующий этап**

Итоговая структура может быть:

* сохранена в results/final.json — **строками по одной записи**;
* добавлена в Excel-файл (если включена экспортная опция);
* передана через API в модуль 1С или neuro\_router.

def save\_final\_record(record: dict, output\_path: str = "results/final.json"):

os.makedirs(os.path.dirname(output\_path), exist\_ok=True)

with open(output\_path, 'a', encoding='utf-8') as f:

f.write(json.dumps(record, ensure\_ascii=False) + '\n')

### 🧩 ****Интеграция:****

| **Модуль** | **Назначение** |
| --- | --- |
| state\_manager.py | Формирует и сохраняет итоговую запись |
| exporter.py | Экспортирует в Excel (если включено) |
| sender.py | Передаёт данные через API (если включено) |

# ****8. 🔄 Загрузка в 1С****

## 🧾 **8.1. Формат данных для API 1С**

### 🎯 ****Назначение:****

Обеспечить корректную, стандартизированную и автоматически обрабатываемую передачу информации из системы ИИ в систему 1С через API-интерфейс. Передача должна быть **однозначной, без потери данных, строго по согласованной структуре**.

### 📌 ****Основные принципы:****

1. **Формат передачи**:  
   – Только JSON, кодировка UTF-8.  
   – Один HTTP POST-запрос передаёт одну запись или массив записей.
2. **Тип структуры**:  
   – Основной объект — **словарь (dict)**.  
   – При передаче массива — **список объектов (list[dict])**.
3. **Именование полей** — **на русском языке**, строго в соответствии с полями, заведёнными в регламенте обмена 1С:
   * допускаются только точные ключи: "Серия ИП", "Дата", "Цель мероприятия" и т.д.;
   * любые сокращения, аббревиатуры или переименования не допускаются.
4. **Типы данных:**

| **Поле** | **Тип данных** | **Примечание** |
| --- | --- | --- |
| Серия ИП | string | Формат: 123456/24/98765-ИП |
| Дата | string (date) | Формат: ДД.ММ.ГГГГ |
| Мероприятие | string | Полное название действия |
| № ИД | string | Совпадает с Серия ИП |
| Адресат мероприятия | string | Значение из справочника |
| Адрес мероприятия | string | Адрес текстом (если есть) |
| Тип мероприятия | string | Например: "Жалоба", "Уведомление" |
| Цель мероприятия | string | Строго по тексту документа |
| Комментарий о типе | `string | null` |
| Комментарий | `string | null` |
| Ответственный | `string | null` |
| Файл | string | Имя обработанного файла |
| Оригинал | string | Путь к исходному файлу |
| Обработано | string (ISO) | Время генерации (формат YYYY-MM-DDTHH:MM:SS) |
| Status | string | Один из: "OK", "неполные данные", "ошибка" |

1. **Обязательные поля** при передаче:
   * Серия ИП, Дата, Тип мероприятия, Цель мероприятия, Адресат мероприятия, Status

Все остальные поля — **опциональны**, но желательны.

1. **Порядок полей не влияет** на обработку в 1С, но желательно придерживаться согласованной структуры, для удобства логирования и отладки.

### 🔒 ****Безопасность и транспорт:****

* Передача осуществляется через защищённый канал HTTPS.
* Используется токен-авторизация или API-ключ из config.yaml.
* Никаких вложенных файлов (PDF, DOCX) в JSON — только текстовая структура.

### 📦 ****Формат готов к приёму в 1С следующими модулями:****

* **Обработчик JSON в 1С**, написанный на платформе 8.3, обрабатывает список записей и загружает в регистры или документы на основании данных;
* Логика проверки обязательных полей реализуется на стороне 1С **повторно**, для двойной гарантии (валидация перед импортом).

## 🧾 **8.2. Алгоритм автоматической загрузки**

### 🎯 ****Назначение:****

Обеспечить **полностью автоматическую, надёжную и контролируемую передачу** подготовленных JSON-записей из нейросетевого парсера в систему 1С через API, с повторными попытками и журналированием на каждом этапе.

### 📌 ****Шаги алгоритма:****

#### 🔹 **Шаг 1. Подготовка JSON-файла**

* Модуль exporter.py формирует файл выгрузка\_YYYYMMDD\_N.json, содержащий **только записи со статусом "OK"**.
* Все записи в файле:
  + прошли валидацию;
  + содержат обязательные поля;
  + нормализованы по структуре из п. 8.1.

#### 🔹 **Шаг 2. Отправка данных через API**

* Используется модуль api\_client.py:
  + Загружает параметры API из config.yaml:
    - url: адрес endpoint в 1С (например, https://1c.local/api/upload\_ip\_data);
    - token или auth\_key: ключ авторизации.
  + Выполняется POST-запрос с телом JSON.

import requests

def send\_to\_1c(json\_data, config):

headers = {

"Authorization": f"Bearer {config['token']}",

"Content-Type": "application/json"

}

response = requests.post(config["url"], json=json\_data, headers=headers)

return response.status\_code, response.text

#### 🔹 **Шаг 3. Контроль ответа**

| **Ответ от 1С** | **Действие** |
| --- | --- |
| 200 OK / 201 Created | Считается успешной загрузкой |
| 4xx, 5xx, ошибка соединения | Считается неудачной попыткой загрузки |

* Результат логируется в process\_log.json:
  + дата, время;
  + имя файла;
  + кол-во строк;
  + статус загрузки;
  + тело ответа (если ошибка).

### 🔁 ****Шаг 4. Повторная отправка****

* Все файлы/строки, не переданные из-за ошибок, фиксируются в error\_log.json и upload\_queue.json.
* Повтор отправки осуществляется:
  + либо при следующем запуске системы;
  + либо по расписанию (например, через cron или встроенный state\_manager.py).
* Каждая повторная попытка сопровождается новой записью в process\_log.json.

### 🛑 ****Особенности контроля:****

* Все ответы от 1С логируются в папке logs/upload/.
* В случае системной ошибки (например, неверный токен, сервер 1С недоступен):
  + файл остаётся в очереди;
  + автоматически не удаляется до успешной передачи;
  + оператор получает уведомление (если включено).

### 🧩 ****Интеграция:****

| **Модуль** | **Назначение** |
| --- | --- |
| exporter.py | Формирует JSON-пакет для отправки |
| api\_client.py | Отправка POST-запроса, приём ответа |
| state\_manager.py | Управление статусами, очередью на повторную |
| process\_log.json | Фиксация успешных и неуспешных загрузок |
| upload\_queue.json | Очередь файлов для повторной отправки |
| error\_log.json | Текст ошибок при отправке |

## 🧾 **8.3. Контроль и прозрачность**

### 🎯 ****Назначение:****

Обеспечить **прозрачный контроль каждого этапа загрузки** данных в 1С: от формирования пакета до получения ответа от API, с возможностью легко определить:

* что загружено;
* что не прошло;
* по каким причинам;
* и что будет отправлено повторно.

### 📋 ****Механизмы контроля:****

#### 🔹 **1. Журнал успешных загрузок —** process\_log.json

Для каждой попытки загрузки фиксируются:

| **Поле** | **Описание** |
| --- | --- |
| datetime | Время попытки загрузки |
| filename | Имя отправленного файла |
| count\_total | Общее число строк в пакете |
| count\_uploaded | Сколько записей успешно загружено |
| status | "ok" или "partial" или "error" |
| api\_response | Ответ от 1С (тело/код) |

📌 Пример записи:

{

"datetime": "2025-06-22T16:45:01",

"filename": "выгрузка\_20250622\_001.json",

"count\_total": 120,

"count\_uploaded": 118,

"status": "partial",

"api\_response": "2 строки не приняты. Причина: неверный формат даты"

}

#### 🔹 **2. Журнал ошибок —** error\_log.json

Фиксируются только записи, **не принятые 1С**, с указанием причины.

📌 Пример строки:

{

"datetime": "2025-06-22T16:45:02",

"record": {

"Серия ИП": "123456/24/98765-ИП",

"Дата": "31.06.2024"

},

"error": "Дата не существует"

}

#### 🔹 **3. Очередь на повтор —** upload\_queue.json

Содержит список файлов или строк, которые необходимо **отправить повторно** при следующем цикле загрузки.

* Записи автоматически переносятся в очередь при ошибке;
* При следующей итерации читаются из очереди и пробуются снова;
* После успешной отправки — удаляются из очереди.

#### 🔹 **4. Контроль оператором**

* Оператор может:
  + открыть логи;
  + выгрузить неудачные строки;
  + вручную инициировать повторную отправку;
* При включённом GUI — выводится статистика по статусам:
  + 🟢 OK: 118 строк;
  + 🔴 Ошибка: 2 строки;
  + 🔁 В очереди: 2 строки.

### ✅ ****Преимущества подхода:****

* **Полная прослеживаемость** всех операций;
* Быстрая диагностика проблем (ошибки на стороне 1С, формат, отсутствие поля);
* Простота повторной отправки и техподдержки;
* Гибкая обработка отказов (частичная загрузка ≠ сбой всей операции).

# ****9. 🗂️ Логирование и аудит****

## **9.1. Ведение журналов обработки и ошибок**

* **process\_log.json**  
  — Основной журнал: по каждой операции (файл, статус обработки, время, результат загрузки в 1С).
* **error\_log.json**  
  — Все ошибки: при чтении, валидации, парсинге, отправке в 1С, сбои соединения, некорректный ответ API.
* **duplicates\_log.json**  
  — Каждый случай дублирования (номер ИП, дата, путь, время попытки загрузки).
* **not\_processed.json**  
  — Всё неразобранное/невалидное (пропущено при обходе, не открывается, не хватает полей).
* **pause.flag**  
  — Текстовый или пустой файл. Его наличие в /logs/ — мгновенно ставит процесс на паузу до удаления файла.

## **9.2. Формат и хранение логов**

* **Формат всех логов — JSON-массив объектов**, где каждая запись структурирована и содержит:
  + дату и время (datetime)
  + этап (stage или event)
  + имя файла/кредитора/должника
  + статус (success, error, skipped, duplicate, и пр.)
  + подробности ошибки (если есть)
  + результат/комментарий

**Пример:**

{

"datetime": "2024-06-20T13:45:00",

"event": "api\_send",

"file": "doc001.pdf",

"creditor": "Сбербанк",

"debtor": "Иванов И.И.",

"status": "error",

"error\_msg": "API 1С: неверный формат даты"

}

* **Все логи и журналы хранятся только в папке /logs/**.  
  — Строгая ротация (например, по датам или объёму), если требуется.

## **9.3. Доступность и аудит**

* Журналы доступны для анализа в любой момент (открываются в любом JSON-редакторе или Python).
* Быстрое восстановление истории операций, поиск проблем, формирование внутренних отчётов.
* Аудит возможен как вручную, так и через модульные отчёты (exporter.py, state\_manager.py).

# ****10. ♻️ Запуск, координация и “сквозной сценарий”****

## **10.1. Общая логика запуска**

* **main.py** — единая точка входа, откуда стартует весь цикл:
  + Проверяет окружение, справочники, доступность директорий.
  + Оркестрирует запуск всех модулей: filewalker → parser → validator → ai\_client → exporter → api\_client.
  + Управляет состоянием (start/stop/pause), реакция на флаги/сигналы, завершение при критических ошибках.

# main.py — Единая точка входа, полный цикл обработки

import os

import sys

import time

from app.config import load\_configs

from app.validator import validate\_all\_configs

from app.state\_manager import (

log\_event,

check\_pause\_flag,

init\_journals,

close\_journals,

)

from app.filewalker import collect\_files

from app.parser import process\_files

from app.ai\_client import analyze\_with\_ai

from app.exporter import export\_to\_json

from app.api\_client import send\_to\_1c\_api

# 1️⃣ Запуск: Проверка окружения и подготовка логов

def main():

try:

# Проверка на наличие pause.flag

if check\_pause\_flag():

print("Pause flag detected — exiting gracefully.")

log\_event(stage="init", status="pause", comment="Process paused by pause.flag")

sys.exit(0)

print("Старт обработки — инициализация журналов и окружения.")

init\_journals()

# 2️⃣ Загрузка и валидация всех справочников и конфигов

print("Загрузка и валидация справочников...")

configs = load\_configs()

validation\_ok, errors = validate\_all\_configs(configs)

if not validation\_ok:

log\_event(stage="init", status="critical\_error", error\_msg=str(errors))

print("Критическая ошибка конфигов! Детали:", errors)

sys.exit(1)

# 3️⃣ Обход файловой структуры: сбор списка файлов к обработке

print("Обход архивов...")

files\_to\_process = collect\_files(configs)

log\_event(stage="filewalker", status="ok", count=len(files\_to\_process))

# 4️⃣ Парсинг и анализ файлов

print(f"Парсинг {len(files\_to\_process)} файлов...")

parsed\_results = process\_files(files\_to\_process, configs)

log\_event(stage="parser", status="ok", count=len(parsed\_results))

# 5️⃣ Интеграция с AI для извлечения смысловых данных

print("Запуск AI-анализа...")

ai\_results = analyze\_with\_ai(parsed\_results, configs)

log\_event(stage="ai\_client", status="ok", count=len(ai\_results))

# 6️⃣ Экспорт результата в JSON для загрузки в 1С

print("Формирование итоговой выгрузки...")

export\_path = export\_to\_json(ai\_results, configs)

log\_event(stage="exporter", status="ok", file=export\_path)

# 7️⃣ Отправка данных в 1С по API

print("Отправка в 1С по API...")

upload\_ok, failed\_rows = send\_to\_1c\_api(export\_path, configs)

if upload\_ok:

log\_event(stage="api\_client", status="ok", file=export\_path)

print("Загрузка завершена успешно!")

else:

log\_event(stage="api\_client", status="error", file=export\_path, details=failed\_rows)

print("Загрузка завершена с ошибками, см. логи.")

# 8️⃣ Финализация

close\_journals()

print("Обработка завершена. Все статусы записаны в журналы.")

except Exception as ex:

log\_event(stage="main", status="exception", error\_msg=str(ex))

print("Ошибка выполнения main.py:", ex)

close\_journals()

sys.exit(2)

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

main()

## **10.2. Порядок запуска и связей между модулями**

1. **Инициализация среды:**
   * Активация venv, загрузка конфигов, логирование старта.
2. **Проверка и загрузка справочников:**
   * 2.1 config.py

import os

import pandas as pd

import yaml

def load\_configs():

configs = {}

# Загружаем CSV-справочники

for fname in ['formats.csv', 'adresat.csv', 'responsible.csv', 'events.csv', 'creditors\_to\_process.csv']:

path = os.path.join('configs', fname)

configs[fname] = pd.read\_csv(path, encoding='utf-8')

# Загружаем YAML-валидаторы

with open(os.path.join('configs', 'validators.yaml'), encoding='utf-8') as f:

configs['validators.yaml'] = yaml.safe\_load(f)

return configs  
,

2.2 validator.py  
  
def validate\_all\_configs(configs):

errors = []

# Проверяем все справочники на наличие, заполненность, отсутствие дублей и др.

for fname, df in configs.items():

if df is None or (hasattr(df, 'empty') and df.empty):

errors.append(f"{fname} пустой или не загружен")

# Примеры валидаторов для CSV:

if fname == "creditors\_to\_process.csv":

if 'creditor' not in df or df['creditor'].duplicated().any():

errors.append("Дубли или нет поля 'creditor' в creditors\_to\_process.csv")

if 'status' not in df or not df['status'].isin(['к обработке', 'пропустить']).all():

errors.append("Некорректный статус в creditors\_to\_process.csv")

# ...дополнительные проверки для других файлов

return (len(errors) == 0), errors  
  
,

2.3 state\_manager.py  
  
import json

from datetime import datetime

def log\_event(\*\*kwargs):

entry = dict(datetime=str(datetime.now()))

entry.update(kwargs)

with open('logs/process\_log.json', 'a', encoding='utf-8') as f:

f.write(json.dumps(entry, ensure\_ascii=False) + "\n")

* + Если что-то не так — аварийный выход.

1. **Обход архивов (filewalker.py):**

import os

import pandas as pd

import json

def load\_processed\_files(log\_path='logs/process\_log.json'):

"""Читает все уже обработанные файлы из process\_log.json"""

processed = set()

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

return processed

with open(log\_path, 'r', encoding='utf-8') as f:

for line in f:

try:

entry = json.loads(line)

if entry.get('status') == 'ok':

processed.add(entry.get('file'))

except Exception:

continue

return processed

def collect\_files(configs):

"""

Обходит архивы, формирует список файлов, которые ещё не обработаны.

— configs['creditors\_to\_process.csv']: где искать

— configs['formats.csv']: какие форматы разрешены

"""

creditors = configs['creditors\_to\_process.csv']

allowed\_exts = set(configs['formats.csv']['extension'].str.lower())

processed\_files = load\_processed\_files()

files\_for\_processing = []

for idx, row in creditors.iterrows():

if row.get('status', '').strip().lower() != 'к обработке':

continue

base\_dir = row['link']

for root, dirs, files in os.walk(base\_dir):

for file in files:

ext = os.path.splitext(file)[-1][1:].lower()

full\_path = os.path.join(root, file)

if ext in allowed\_exts and full\_path not in processed\_files:

files\_for\_processing.append({

'creditor': row['creditor'],

'file': full\_path,

'ext': ext

})

return files\_for\_processing

* + Получение и фильтрация списка файлов для обработки.
  + Логика обхода учитывает уже обработанные/пропущенные файлы.

1. **Парсинг и анализ файлов (parser.py, validator.py):**

**parser.py**import os

from app.validator import validate\_document\_fields

from app.state\_manager import log\_event

def parse\_file(file\_info, configs):

"""

file\_info — dict (creditor, file, ext)

configs — справочники для поиска по полям

Возвращает dict с извлечёнными полями или None

"""

try:

# 1. Попытка открытия файла

if not os.path.exists(file\_info['file']):

log\_event(stage="parser", status="error", file=file\_info['file'], error\_msg="File not found")

return None

# 2. Чтение и первичный парсинг (PDF, DOCX...)

text = extract\_text(file\_info['file'], file\_info['ext']) # реализовать под свой формат

# 3. Извлечение данных (примерно, можно усложнять)

doc\_data = extract\_fields\_from\_text(text, configs)

# 4. Валидация полей (валидаторы по регуляркам, справочники)

valid, validation\_errors = validate\_document\_fields(doc\_data, configs)

if not valid:

log\_event(stage="parser", status="error", file=file\_info['file'], error\_msg=f"Validation failed: {validation\_errors}")

return None

# 5. Логируем успех

log\_event(stage="parser", status="ok", file=file\_info['file'], result="parsed")

return doc\_data

except Exception as ex:

log\_event(stage="parser", status="error", file=file\_info['file'], error\_msg=str(ex))

return None

def process\_files(files\_to\_process, configs):

"""

Проходит по списку файлов, парсит и валидирует каждый.

Возвращает список успешно распарсенных dict'ов.

"""

parsed = []

for file\_info in files\_to\_process:

doc\_data = parse\_file(file\_info, configs)

if doc\_data:

parsed.append(doc\_data)

return parsed

# ----- Примерные заглушки для функций -----

def extract\_text(path, ext):

# Здесь должна быть логика чтения PDF, DOCX и т.д.

# Например, с помощью pdfminer, docx2txt и др.

# Вернёт сырой текст документа

return "Сырой текст документа"

def extract\_fields\_from\_text(text, configs):

# Здесь распознаются даты, номера, адресаты и т.д.

# Использует регулярки, справочники и т.п.

return {

"date": "20.06.2024",

"number\_ip": "123456/22/12345-ИП",

"adresat": "ФССП",

"type\_event": "Жалоба",

"goal": "Взыскать сумму по ИП",

"comment": "Жалоба на действия пристава",

"responsible": "Иванов И.И.",

"status": "OK"

}  
  
 **validator.py**import re

def validate\_document\_fields(doc, configs):

errors = []

# Пример: дата по шаблону из validators.yaml

date\_regex = configs['validators.yaml']['date']['regex']

if not re.match(date\_regex, doc.get("date", "")):

errors.append("Некорректная дата")

# Пример: обязательные поля

for f in ("number\_ip", "adresat", "type\_event"):

if not doc.get(f):

errors.append(f"Нет поля {f}")

# Пример: номер ИП по регулярке

ip\_regex = configs['validators.yaml']['number\_ip']['regex']

if not re.match(ip\_regex, doc.get("number\_ip", "")):

errors.append("Неверный формат номера ИП")

# Можно добавить проверки по справочникам и т.д.

return (len(errors) == 0), errors

* + Для каждого файла — попытка анализа, логирование результата.
  + Для каждого этапа — обработка ошибок и пропусков.

1. **Интеграция с внешними AI/ML (ai\_client.py):**

import requests

from app.state\_manager import log\_event

YANDEX\_API\_URL = "https://llm.api.yandex.net/your-endpoint" # пример

def analyze\_with\_ai(parsed\_docs, configs):

"""

На вход — список распарсенных документов (dict), на выход — обновлённый список dict с дополненными полями от AI.

"""

ai\_enriched = []

api\_key = configs.get('config.yaml', {}).get('yandex\_api\_key', '')

for doc in parsed\_docs:

try:

# Формируем prompt для AI

prompt = build\_prompt(doc)

# Запрос к внешнему AI (пример с Яндекс GPT/LLM API)

response = requests.post(

YANDEX\_API\_URL,

headers={"Authorization": f"Api-Key {api\_key}"},

json={"text": prompt}

)

if response.status\_code == 200:

ai\_data = response.json()

# Пример: дополняем doc новыми полями

doc['goal'] = ai\_data.get('goal\_short') # краткая цель ("прошу...")

doc['comment'] = ai\_data.get('summary') # краткий комментарий/резюме

doc['ai\_status'] = 'OK'

log\_event(stage="ai\_client", status="ok", file=doc.get("file"), result="AI enriched")

else:

doc['ai\_status'] = 'error'

doc['ai\_error\_msg'] = f"HTTP {response.status\_code}: {response.text}"

log\_event(stage="ai\_client", status="error", file=doc.get("file"), error\_msg=doc['ai\_error\_msg'])

except Exception as ex:

doc['ai\_status'] = 'error'

doc['ai\_error\_msg'] = str(ex)

log\_event(stage="ai\_client", status="error", file=doc.get("file"), error\_msg=str(ex))

ai\_enriched.append(doc)

return ai

* + Отправка текстов в AI для выделения смысловых сущностей/резюме.
  + Получение ответа, валидация результатов.

1. **Формирование выгрузки (exporter.py):**

import os

import json

from datetime import datetime

def export\_to\_json(ai\_results, configs):

"""

ai\_results — список dict с полями по каждому обработанному документу.

Сохраняет все валидные записи в итоговый JSON (без дублей, с флагами статуса).

Возвращает путь к итоговому файлу.

"""

# Проверка на дубли и полноту

seen\_keys = set()

clean\_results = []

for doc in ai\_results:

key = (doc.get('number\_ip'), doc.get('date'))

# Не добавляем дубли

if key in seen\_keys:

continue

seen\_keys.add(key)

# Проверка на полноту — если чего-то не хватает, ставим статус

if not doc.get('number\_ip') or not doc.get('date') or not doc.get('type\_event'):

doc['status'] = 'неполные данные'

elif doc.get('ai\_status') == 'error':

doc['status'] = 'ошибка'

else:

doc['status'] = 'OK'

clean\_results.append(doc)

# Имя файла — по дате и порядковому номеру

export\_dir = "exports"

os.makedirs(export\_dir, exist\_ok=True)

fname = f"выгрузка\_{datetime.now().strftime('%Y%m%d\_%H%M%S')}.json"

export\_path = os.path.join(export\_dir, fname)

# Сохраняем

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

json.dump(clean\_results, f, ensure\_ascii=False, indent=2)

return export\_path

* + Сбор всех валидных результатов в итоговую JSON-структуру.
  + Проверка на дубли и полноту данных.

1. **Загрузка в 1С (api\_client.py):**
   * Автоматическая отправка по API.
   * Проверка статусов загрузки, повторная отправка при ошибках.
2. **Ведение журналов (state\_manager.py):**

import os

import json

from datetime import datetime

LOG\_DIR = "logs"

os.makedirs(LOG\_DIR, exist\_ok=True)

def \_write\_log(filename, entry):

path = os.path.join(LOG\_DIR, filename)

with open(path, 'a', encoding='utf-8') as f:

f.write(json.dumps(entry, ensure\_ascii=False) + "\n")

def log\_event(\*\*kwargs):

"""Логирует стандартные события (успехи и статусы этапов)"""

entry = dict(datetime=str(datetime.now()), \*\*kwargs)

\_write\_log("process\_log.json", entry)

def log\_error(\*\*kwargs):

"""Логирует все ошибки: чтение, парсинг, API, валидация"""

entry = dict(datetime=str(datetime.now()), \*\*kwargs)

\_write\_log("error\_log.json", entry)

def log\_duplicate(file, number\_ip, date):

"""Логирует все дубли (номер ИП, дата, путь к файлу)"""

entry = {

"datetime": str(datetime.now()),

"file": file,

"number\_ip": number\_ip,

"date": date,

"event": "duplicate"

}

\_write\_log("duplicates\_log.json", entry)

def log\_not\_processed(file, reason):

"""Логирует всё неразобранное и невалидное"""

entry = {

"datetime": str(datetime.now()),

"file": file,

"reason": reason,

"event": "not\_processed"

}

\_write\_log("not\_processed.json", entry)

def check\_pause\_flag():

"""Проверяет наличие pause.flag для экстренной остановки"""

return os.path.exists(os.path.join(LOG\_DIR, "pause.flag"))

def init\_journals():

"""(опционально) Создаёт пустые журналы при первом запуске"""

for name in ["process\_log.json", "error\_log.json", "duplicates\_log.json", "not\_processed.json"]:

path = os.path.join(LOG\_DIR, name)

if not os.path.exists(path):

with open(path, 'w', encoding='utf-8') as f:

pass

def close\_journals():

"""Заглушка — если нужно делать доп. действия при завершении"""

pass

* + Логирование всего процесса: действий, ошибок, дублей, неразобранных.
  + Отдельные логи для аудита.

1. **Аналитика и мониторинг (exporter.py/reporting, cli.py):**

**import json**

**import os**

**from collections import Counter**

**def generate\_report():**

**"""Генерирует краткий статистический отчёт на основе логов."""**

**logs\_dir = "logs"**

**summary = {}**

**# Анализ общего лога процесса**

**process\_log\_path = os.path.join(logs\_dir, "process\_log.json")**

**statuses = Counter()**

**if os.path.exists(process\_log\_path):**

**with open(process\_log\_path, encoding="utf-8") as f:**

**for line in f:**

**rec = json.loads(line)**

**statuses[rec.get("status", "unknown")] += 1**

**summary['Обработка файлов (process\_log.json)'] = dict(statuses)**

**# Анализ ошибок**

**error\_log\_path = os.path.join(logs\_dir, "error\_log.json")**

**error\_types = Counter()**

**if os.path.exists(error\_log\_path):**

**with open(error\_log\_path, encoding="utf-8") as f:**

**for line in f:**

**rec = json.loads(line)**

**error\_types[rec.get("stage", "unknown")] += 1**

**summary['Ошибки (error\_log.json)'] = dict(error\_types)**

**# Дубли**

**dups\_path = os.path.join(logs\_dir, "duplicates\_log.json")**

**dups\_count = 0**

**if os.path.exists(dups\_path):**

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

**for \_ in f:**

**dups\_count += 1**

**summary['Дублирующихся документов'] = dups\_count**

**# Необработанные**

**notproc\_path = os.path.join(logs\_dir, "not\_processed.json")**

**notproc\_count = 0**

**if os.path.exists(notproc\_path):**

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

**for \_ in f:**

**notproc\_count += 1**

**summary['Необработанных файлов'] = notproc\_count**

**# Вывод отчёта в консоль (или сохранить в файл)**

**print("===== Сводная статистика по логам =====")**

**for k, v in summary.items():**

**print(f"{k}: {v}")**

**# (Опционально) Сохраняем как отдельный JSON-отчёт**

**with open(os.path.join(logs\_dir, "summary\_report.json"), "w", encoding="utf-8") as f:**

**json.dump(summary, f, ensure\_ascii=False, indent=2)**

**# ----- CLI-интерфейс (опционально) -----**

**# app/cli.py**

**def run\_cli():**

**print("1. Сгенерировать отчёт")**

**print("2. Посмотреть ошибки")**

**choice = input("Выберите действие: ")**

**if choice == "1":**

**generate\_report()**

**elif choice == "2":**

**with open("logs/error\_log.json", encoding="utf-8") as f:**

**for line in f:**

**print(line)**

**else:**

**print("Неизвестная команда.")**

* + Отчёты по этапам, генерация статистики, контроль ошибок.

## **10.3. Логические зависимости и события**

* Каждый модуль получает только необходимые данные от предыдущего (минимум глобального состояния).
* Любое отклонение (ошибка, сбой API, невалидность) — мгновенно логируется, остальной процесс не останавливается (fail-tolerant).
* Все ключевые статусы передаются через state\_manager (единый контроль).
* Если срабатывает флаг /logs/pause.flag — все процессы корректно завершают текущий шаг и останавливаются.

## **10.4. Чек-лист готовности сквозного запуска**

* venv активирован, все зависимости установлены
* Все справочники/конфиги доступны и валидны
* Архивы доступны на чтение, выходные папки — на запись
* Журналы не заблокированы, доступны для записи
* main.py — запускается из-под ограниченного пользователя/службы
* Настроена интеграция с API 1С (ключи, адрес)
* Оповещения об ошибках подключены (если нужно)
* Документация и инструкции — в актуальной версии

## **10.5. Сценарии запуска**

* **Запуск вручную:**  
  — Активировать venv  
  — python main.py
* **Автозапуск по расписанию:**  
  — systemd, Task Scheduler, cron, etc.
* **Реакция на паузу/остановку:**  
  — main.py проверяет наличие pause.flag, корректно завершает процесс.