ChatGPT

Blackletter GDPR Vendor Contract Checker

Part 1: Technical Build Guide

Overview

The Blackletter GDPR Vendor Contract Checker is an end-to-end system for automating GDPR compliance reviews of vendor contracts. It ingests contract documents (PDF/DOCX), identifies required GDPR clauses, flags vague terms, and produces a compliance report with remediation suggestions. The architecture is designed as a full-stack MVP with privacy-by-design defaults – processing data in isolated chunks, minimizing sensitive data exposure, and honoring deletion requests. Key components include a modern **frontend** for file upload and results display, a lightweight **API backend** (BFF) to orchestrate tasks, an **ingestion pipeline** for document parsing and OCR, a **GDPR rule engine** with configurable rules (JSON/YAML), a **retrieval-augmented generation (RAG) index** for semantic search, an **LLM-based judge** to assess compliance (with source citation), a **report builder** for outputting HTML/PDF summaries, and background **worker processes** to handle heavy tasks. All data is stored in a secure manner using **PostgreSQL** (with vector search via pgvector) and **S3-compatible storage**, with instrumentation via **OpenTelemetry** and secrets management via **Vault**.

Tech Stack Summary:

- **Frontend:** Next.js (React) with Tailwind CSS and Shadcn/UI components for styling, and TipTap for rich text/annotation (to highlight contract clauses).
- **Backend API:** FastAPI (Python) – or NestJS (TypeScript) as an alternative – acting as a BFF (Backend-For-Frontend) to handle HTTP requests, file uploads, and trigger background workflows.
- **Document Processing:** PDF/DOCX text extraction with libraries (e.g. PyMuPDF, python-docx) and OCR via Tesseract (or AWS Textract for higher accuracy) to handle scanned documents.
- **Rule Engine:** A set of JSON/YAML-defined rules for GDPR clauses (e.g. breach notification, data deletion) and regex patterns for vague terms (“reasonable”, “appropriate”) to flag compliance gaps.
- **Vector Index (RAG):** Document text is chunked (e.g. ~500 tokens per chunk) and embedded using OpenAI or open-source models. Embeddings are stored in a vector database (Postgres pgvector or OpenSearch) to enable hybrid semantic + keyword retrieval of relevant contract clauses [1] [2].
- **LLM Judge:** A large language model (e.g. GPT-4 via OpenAI API) that receives relevant text chunks and questions about GDPR requirements, then determines compliance with *inline citations* referencing the contract text [3]. Prompting is structured to enforce source-backed answers and avoid hallucinations.
- **Report Builder:** Aggregates rule findings and LLM analyses into a final report (HTML for web, PDF for download) that summarizes compliance status and recommended remediations. Also optionally generates a redlined contract (DOCX) highlighting necessary changes [4].
- **Workers & Async:** Heavy tasks (parsing, embedding, LLM calls) are offloaded to background workers (Celery with a Redis broker, or Temporal workflows) to keep the API responsive [1]. The API immediately returns a job ID for frontend to poll or subscribe for results.
- **Persistence:** PostgreSQL stores structured data (users, jobs, results, rules) and vector embeddings (via pgvector extension) to enable similarity search [5]. An S3-compatible object

storage (e.g. AWS S3, MinIO) holds uploaded files and generated reports. All data is encrypted at rest and scoped per tenant for isolation.
- **Observability & Security:** The system is instrumented with OpenTelemetry for end-to-end tracing and logging. Secrets (API keys, DB passwords) are stored in Vault or environment variables, never hard-coded. Privacy-by-design measures include optional PII redaction before LLM processing, processing data in minimal chunks, and automatic data retention limits (e.g. 30-day deletion by default [6]). Audit logs record document access and processing events for compliance.

Below, we provide a step-by-step guide through each layer of the stack, with example configurations and code references, followed by a visual architecture diagram.

Frontend – Next.js, TipTap, Tailwind, Shadcn/UI

The frontend is built with **Next.js 13+ (React)**, using the App Router for a modern, file-based routing structure. It provides an intuitive UI for users to upload contracts, monitor processing status, and view the compliance report. Key technologies include **Tailwind CSS** for styling, **shadcn/ui** (a library of pre-built Tailwind components) for consistent UI elements, and **TipTap** for rich text display and annotation of contract text.

Project Structure: In a Next.js app directory setup, we might organize as follows:

```
frontend/
├── app/
│   ├── upload/page.tsx         # Upload page for contract review
│   ├── report/[id]/page.tsx    # Report page to display results for a given
job
│   └── ... (other routes, e.g., auth)
├── components/
│   ├── UploadDropzone.tsx      # Reusable component for file drag-and-drop
upload
│   ├── ReportViewer.tsx        # Component to render report (could use
TipTap for formatting)
│   └── ... (UI components)
├── lib/
│   └── api.ts                  # API client (e.g. Axios or fetch wrapper)
for calling backend
├── styles/ (Tailwind CSS setup)
└── ...
```

The **Upload page** (`app/upload/page.tsx`) provides a drag-and-drop file uploader and form options (if any). For example:

```
// app/upload/page.tsx
import UploadDropzone from '@/components/UploadDropzone';

export default function UploadPage() {
  return (
    <section className="space-y-4">
```

```
      <h2 className="text-2xl font-semibold">Contract Review</h2>
      <UploadDropzone /> {/* Custom dropzone component using shadcn UI
styling */}
    </section>
  );
}
```

When a file is selected, `UploadDropzone` can upload it via the BFF API (using a route like `POST /review`) and then redirect to a status or report page. This component might use a library like **react-dropzone** internally, styled with Tailwind and shadcn/ui (e.g., using a Card or Button component from the library). The UI provides feedback (upload progress, errors) and might immediately show a "Processing…" state after upload, polling the backend for job completion.

Rich Text Display: Once the analysis is done, the report page (`app/report/[id]/page.tsx`) fetches the results (e.g., via `GET /review/{id}`) and displays them. TipTap can be used here to render contract excerpts with highlights or **tooltips**. For instance, if the LLM flagged a clause or a vague term, TipTap's editor can mark those text segments in red or with an underline. This allows users to click or hover and see an explanation (using shadcn/ui Tooltip or Dialog for the popup). For example, TipTap's JSON content model could represent the contract text with marks for issues, and the frontend would map those to tooltips saying *"! This clause is missing a 72-hour breach notification requirement."*.

Styling and Components: Using Tailwind CSS utility classes ensures a consistent design. Shadcn/UI provides ready-made components like **Button**, **Input**, **Toast notifications**, **Progress bars**, etc., which speed up development. For example, a **Progress bar** might show analysis progress if the app receives updates. Shadcn components are built on Radix UI primitives and integrate with Tailwind, so they are highly customizable. The overall design aims to be clean and professional, instilling trust (important for legal tech).

Frontend API Calls: The Next.js app will communicate with the backend BFF via REST (or could use GraphQL, but REST is simpler here). We configure the base URL in an environment variable (e.g. `NEXT_PUBLIC_API_URL`) so that the frontend knows where to send requests. We might create a small `api.ts` library:

```
// lib/api.ts (using fetch API for simplicity)
export const api = {
  get: async (path: string) => {
    const res = await fetch(`${process.env.NEXT_PUBLIC_API_URL}${path}`);
    if (!res.ok) throw new Error(await res.text());
    return res.json();
  },
  post: async (path: string, body: any) => {
    const res = await fetch(`${process.env.NEXT_PUBLIC_API_URL}${path}`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(body),
    });
    if (!res.ok) throw new Error(await res.text());
    return res.json();
  }
};
```

Using this, pages can call `api.post('/review', {...})` to upload a contract (though for file upload specifically, we'd use a `<form>` or FormData via fetch, not JSON). In Next.js, server-side code (if any) could also proxy or handle uploads, but since we have a dedicated BFF, the Next app remains mostly static and purely calls the BFF endpoints.

Tip: During development, set `NEXT_PUBLIC_API_URL="http://localhost:8000"` (or appropriate) so that next and backend communicate. In production, this might be an absolute URL or a relative path if deployed on the same domain.

Backend API – FastAPI/NestJS BFF

The backend serves as a **Backend-for-Frontend (BFF)**, exposing a simple API for the frontend to interact with the system. You can implement this in **FastAPI (Python)** or **NestJS (Node/TypeScript)** – both are suitable. We'll illustrate with FastAPI for concreteness, given its straightforward integration with Python-based processing libraries.

Responsibilities: The BFF handles incoming requests, performs lightweight validation/authentication, and delegates heavy processing to the background workers. It typically will: accept the uploaded file, store it, create a database record for the job, enqueue a background task (passing the file location and job details), and return a response (often containing a `job_id` or similar). It may also provide endpoints to query status or retrieve the final report.

FastAPI Setup: A possible project layout for the backend:

```
backend/
├── main.py                  # FastAPI app initialization, including router
inclusion
├── routers/
│   └── contracts.py         # Defines /review endpoints for contract
checking
├── workers.py               # Celery worker initialization (if using Celery)
├── services/
│   ├── ingest.py            # Functions for parsing and OCR
│   ├── rules_engine.py      # Functions to apply rule checks
│   ├── rag.py               # Functions for chunking and embedding
│   ├── llm_judge.py         # Functions to query LLM
│   └── report_builder.py    # Functions to compile the report
├── models/
│   ├── schemas.py           # Pydantic models (e.g., ReviewResult)
│   └── db.py                # SQLAlchemy models or DB connection setup
├── rules/
│   └── gdpr_playbook.yaml   # Default GDPR rules definitions
└── config.py                # Configuration (e.g., constants, feature flags)
```

In `main.py`, we create the FastAPI app, include routers, and possibly add middleware (for CORS, telemetry, etc.):

```python
# main.py
import os, uvicorn
from fastapi import FastAPI
from routers import contracts

app = FastAPI(title="GDPR Contract Checker API")
app.include_router(contracts.router)

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Review Endpoint: The core endpoint might look like this (from `routers/contracts.py`):

```python
from fastapi import APIRouter, UploadFile, File, Form, BackgroundTasks,
HTTPException
from models.schemas import ReviewResult
from workers import process_contract  # Celery task or function

router = APIRouter()

@router.post("/review", response_model=ReviewResult)
async def review_contract(background_tasks: BackgroundTasks,
                          file: UploadFile = File(...),
                          contract_type: str = Form("vendor_dpa"),
                          jurisdiction: str = Form("EU"),
                          playbook_id: str | None = Form(None)):
    # 1. Save uploaded file to storage
    contents = await file.read()
    file_path = f"/tmp/{file.filename}"
    with open(file_path, "wb") as f:
        f.write(contents)
    # 2. Create DB record for this review job (omitted for brevity)
    job_id = create_job_record(file_path, contract_type, jurisdiction,
playbook_id)
    # 3. Launch background task for processing
    background_tasks.add_task(process_contract, job_id)
    # 4. Return initial response
    return {"job_id": job_id, "status": "queued"}
```

This demonstrates an async endpoint that receives a file and form fields (like contract type or jurisdiction, which could tailor the rule playbook). It saves the file (here to local `/tmp`, but in practice to S3 or a persistent volume), records a job in the database, then uses FastAPI's `BackgroundTasks` to call the `process_contract` task. In production, you'd likely dispatch via Celery instead (e.g., `process_contract.delay(job_id)` if `process_contract` is a Celery task). The response immediately returns a `job_id` and status.

Alternate (NestJS): If using NestJS, the flow is analogous: create a controller with a POST endpoint, use `FileInterceptor` to handle file upload (multipart form data), save the file, and then push a message to a queue (e.g., using a BullJS queue or calling an external worker service). The overall architecture remains the same regardless of framework.

API Responses: The API should also provide a way to fetch results. For example, a GET `/review/{job_id}` returning `ReviewResult` (a Pydantic model or DTO) which includes summary, list of issues found, etc. This allows the frontend to poll the status. Alternatively, one could incorporate WebSockets or Server-Sent Events to push status updates (this is an enhancement for real-time feedback, though a simple polling loop is acceptable for MVP).

Data Models: Using Pydantic (FastAPI) or DTOs (NestJS) ensures well-defined input and output schemas. For instance, a `ReviewResult` model might be:

```
class Issue(BaseModel):
    rule_id: str
    description: str
    compliant: bool
    details: str | None = None    # additional info or excerpt
    citation: str | None = None   # reference to contract text if applicable

class ReviewResult(BaseModel):
    summary: str
    issues: list[Issue]
    remediation: list[str]        # recommended fixes or actions
    report_url: str | None = None # link to PDF report if available
    status: str
```

During processing, the backend can populate this model. Initially, `status` might be "queued" or "processing", and later "completed". The client could retrieve this JSON and render accordingly. The citation field in each issue is meant to hold a reference (like a snippet or a pointer to where in the contract the evidence is), reinforcing traceability.

Tenant and Auth: If this is a multi-tenant SaaS, the API should authenticate users (e.g., via a JWT or session) and associate each job with a user/tenant ID. For simplicity, one could integrate Auth0/Clerk, or even use a Supabase Auth if using Supabase as the backend. At minimum, ensure that one client cannot access another's `job_id` results – e.g., by scoping queries to the authenticated user.

Document Ingestion – PDF/DOCX Parsing & OCR

Once a contract file is uploaded, the first background step is **document ingestion**, which entails extracting text from the document. The system must handle a variety of inputs: machine-readable PDFs, scanned/image PDFs, Microsoft Word documents, etc. We use open-source libraries and OCR to get the text content needed for analysis.

Parsing PDFs: For PDFs that contain embedded text (e.g. not scans), libraries like **PyMuPDF (fitz)** or **pdfplumber** can directly extract text from each page. These libraries preserve basic layout and are efficient. Example using PyMuPDF:

```python
import fitz  # PyMuPDF
def extract_text_from_pdf(path):
    text = ""
    doc = fitz.open(path)
    for page in doc:
        text += page.get_text()  # extract text of the page
    return text
```

This yields a raw text string. We might split it into pages or sections (e.g., using form feed or page number markers) if needed. Another option is **unstructured** (from Unstructured.io) which can parse documents and output text with minimal formatting – useful if we need structured elements like headings or lists, but for our purposes raw text is often enough.

Parsing Word Documents: For DOCX files, use **python-docx** (python library) or **Mammoth** (JS library) if in a Node environment. Python-docx can read text paragraphs easily:

```python
from docx import Document
def extract_text_from_docx(path):
    doc = Document(path)
    return "\n".join(p.text for p in doc.paragraphs)
```

We might also capture headings or table text if relevant. But since GDPR clauses are usually in paragraph form, a simple concatenation of all text works. Keep in mind that formatting (like numbered clauses) might be lost; if needed, we can iterate through runs or preserve section headings to help identify clause boundaries.

OCR for Scanned Docs: If the PDF is actually scanned images (no extractable text), we integrate **OCR**. For cost efficiency, we use **Tesseract OCR** via **pytesseract**. We can detect if a PDF page has text by checking extraction output or using PyMuPDF's `page.get_text()`. If blank or very sparse, it likely needs OCR. A pipeline for OCR might be:

1. Convert each PDF page to an image. Use something like `fitz` (which can render page to image) or **pdf2image** to get a PIL image of each page.
2. Run `pytesseract.image_to_string(image)` on each image.
3. Collect the text, perhaps tagging it with the page number.

Example:

```python
import pytesseract
from pdf2image import convert_from_path

def ocr_pdf(path):
    images = convert_from_path(path)
    text_pages = []
    for i, img in enumerate(images, start=1):
        txt = pytesseract.image_to_string(img)
        text_pages.append(f"[[Page {i}]]\n{txt}")
    return "\n".join(text_pages)
```

Tesseract is free but can be slow on large documents; for better accuracy on forms or dense text, one could opt for **AWS Textract** or **Google Vision OCR**, which have superior OCR (especially for handwritten or tabular data) but incur API costs. Given the budget constraints, Tesseract is preferred unless the image quality is poor.

Preprocessing & Redaction: As part of ingestion, we can optionally perform **redaction** of personally identifiable information (PII) or other sensitive data. For example, if the contract has names, addresses, or contact info, we might mask those before sending text to external LLM APIs. This could be as simple as regex patterns for emails, phone numbers, or using a small PII detection library (or even an AI model) to find and replace such info with placeholders (e.g., "ACME Corp" -> "[Vendor]"). This ensures we uphold data minimization – only the necessary contract wording is exposed for analysis. Redaction is especially relevant if using third-party LLM APIs (OpenAI) to avoid leaking sensitive personal data.

Clause Segmentation (Optional): It might be helpful to split the text by clause or section at this stage. For instance, look for headings or numbered paragraphs. The **rule engine** (next step) could benefit from knowing clause boundaries (e.g., separate sections like "Security", "Sub-processing", etc.). If the contract is well-structured (many are), we could split on numbered list items or keywords like "Clause", "Section", or even detect if lines end with semicolons etc. However, a simpler approach is to treat the whole text (or page by page) for scanning, and rely on search/embeddings later to find specific content.

After extraction, the raw text (or structured text) is stored or passed along to the next steps. We might save it in the database (for record-keeping and for vectorization) or keep it in memory through the task chain. Storing the full text in the DB (in a TEXT column) under the job record allows easy retrieval for debugging or re-processing without original file, but it's optional.

GDPR Rule Engine – JSON Rules & Regex Checks

Before (and alongside) using AI, the system employs a **rule-based engine** to perform deterministic checks on the contract text. This rule engine encodes known GDPR requirements for vendor contracts (often called a Data Processing Agreement or DPA) and scans the text for their presence or absence. It also flags **vague terms** that could weaken the contract.

Rules Definition: Rules can be defined in a **JSON** or **YAML** file (often called a "playbook"). Each rule might include: an `id`, a description of the GDPR clause, a regex pattern or keyword list to search for, and the logic for pass/fail. For example, a YAML rule set might look like:

```yaml
# rules/gdpr_playbook.yaml
- id: breach_notification
  description: "Breach notification within 72 hours"
  pattern: "(breach|incident).*72\s*hours"
  required: true
  if_missing: "No clause requiring breach notification within 72 hours was
found."
  if_found: "Includes a breach notification clause."
- id: data_deletion
  description: "Deletion or return of data on contract end"
  pattern: "delete.*(end|termination)|return.*(end|termination)"
  required: true
  if_missing: "No clause for deletion or return of personal data upon
termination."
  if_found: "Includes end-of-contract data deletion/return clause."
- id: vague_terms
  description: "Avoid vague terms (\"reasonable\", etc.)"
  pattern: "\\b(reasonable|appropriate|best effort)\\b"
  required: false  # not mandatory clause, but flag usage
  warning: "Uses vague term '{match}' which could be made more specific."
  if_found: "Vague term '{match}' detected in contract."
```

Each rule contains a regex `pattern` to search in the text. For required clauses (like breach notification, data deletion, sub-processor consent, etc.), if the pattern is not found (`if_missing`), we flag a compliance gap. The rule descriptions are rooted in GDPR Article 28(3), which mandates specific clauses in vendor contracts (e.g. breach notification, assistance with data subjects' rights, use of sub-processors only with consent, etc.) [7]. The ICO guidance, for example, lists *"the contract must include clauses on processing only on instructions, confidentiality, security measures, sub-processor conditions, assisting the controller, end-of-contract provisions, etc."* [7] – our rules cover these points.

Regex Matching: The engine will run each regex on the contract text (case-insensitive matching, and perhaps with `re.DOTALL` if patterns span lines). If a match is found, the rule passes (or triggers a warning), possibly capturing the exact phrase (`{match}`) for detailed reporting. If not found and the clause is `required`, it's a failure – the contract is missing this required term.

Example: For the *breach_notification* rule, the regex might find phrases like "notify the Controller **without undue delay** and in any event within **72 hours** of becoming aware of a personal data breach". If found, we mark it compliant (maybe even store the snippet). If not, it's non-compliant and we'll later prompt the LLM to confirm or elaborate.

Vague Terms: The *vague_terms* rule above scans for words like "reasonable" or "appropriate". These aren't compliance requirements per se, but they indicate weak language. If found, we flag them as issues to potentially revise (the output can say e.g., "uses vague term 'reasonable' in security measures clause – consider defining specific requirements"). This uses the `{match}` substitution to list the actual vague word found.

Rule Engine Execution: Implementation-wise, we can load the YAML into a Python list of dicts (using PyYAML or similar). Then:

```python
import re, yaml

with open('rules/gdpr_playbook.yaml') as f:
    rules = yaml.safe_load(f)

def run_rules_on_text(text: str) -> list[Issue]:
    issues = []
    for rule in rules:
        pattern = re.compile(rule['pattern'], flags=re.IGNORECASE|re.DOTALL)
        matches = pattern.findall(text)
        if rule.get('required', False):
            if matches:
                # Clause found
                issues.append(Issue(rule_id=rule['id'],
description=rule['description'],
                                    compliant=True,
details=rule.get('if_found')))
            else:
                # Clause missing
                issues.append(Issue(rule_id=rule['id'],
description=rule['description'],
                                    compliant=False,
details=rule.get('if_missing')))
        else:
            # not strictly required (could be a warning)
            if matches:
                # e.g., vague term found – report a warning issue
                found = matches[0] if isinstance(matches, list) else matches
                details = rule.get('warning', rule.get('if_found',
'')).replace("{match}", found)
                issues.append(Issue(rule_id=rule['id'],
description=rule['description'],
                                    compliant=False, details=details))
    return issues
```

Here `Issue` could be a Pydantic model or simple dataclass. We treat non-required rules as warnings (mark `compliant=False` but perhaps differentiate in UI). The regex can capture the first instance of the vague term for reporting; if multiple occurrences, we might list them all or just note the presence.

Integration with LLM: The rule engine provides quick wins – if a required clause is entirely missing, we already know it's a problem and can note it. If a clause is present, we know where (via the regex match index or simply that it exists). However, just presence doesn't guarantee adequacy. For example, the contract might mention “breach notification” but say “notify in 30 days” which is not compliant with GDPR's 72-hour expectation. A regex can't easily catch that nuance. This is where the **LLM judge** comes in. We will use the results of the rule engine to guide the LLM: e.g., if a rule's pattern is found, feed that clause to the LLM to verify if it meets the standard; if not found, possibly ask the LLM “confirm that no clause about X exists and if that's a problem”.

By structuring rules in a data file, non-developers (or legal experts) can adjust the playbook as regulations evolve. The YAML/JSON is essentially a *knowledge base of GDPR requirements*. We might allow multiple playbooks (the `playbook_id` parameter in the API) – for instance, a UK-specific one or sector-specific tweaks. The system can load the selected YAML at runtime to apply those checks.

RAG Indexer – Chunking, Embeddings & Hybrid Search

To effectively utilize the LLM and keep context windows small, the system employs **Retrieval-Augmented Generation (RAG)** techniques. This involves breaking the contract into chunks, embedding those chunks, and performing semantic search to fetch relevant text for each query. The goal is to **only send pertinent snippets to the LLM**, preserving privacy (no need to expose the whole contract) and reducing token usage.

Text Chunking: After text extraction, we split the contract into semi-structured chunks. A chunk could be a paragraph, a clause, or a fixed token window (e.g. 300-500 words or ~500 tokens). It's useful to align chunks with logical sections (e.g., by splitting on double newlines, headings, or clause separators) so that each chunk is a coherent piece of text. For example, we can split on newline patterns or using an NLP library to split by sentences until a token limit is reached. An overlapping window technique (like sliding window with 50% overlap) can help maintain context on boundaries. In practice, for legal text, splitting by sections (if headings or numbering exist) is ideal – e.g., each numbered clause becomes a chunk.

Embedding Model: Each chunk is converted to a vector embedding. We can use OpenAI's embedding API (e.g. `text-embedding-ada-002`) or a local model from **Sentence Transformers** (like `all-MiniLM-L6-v2`). OpenAI's embeddings are high-quality and cost-effective (~$0.0004 per 1K tokens), and with small documents the cost is negligible. Alternatively, to keep everything self-hosted, sentence-transformer models can be loaded (they're a few hundred MB at most) and run on CPU. For an MVP with limited budget, one approach is to start with OpenAI embeddings for simplicity, then move to an open model if usage grows.

The embedding step produces a vector (e.g., 1536-dim for OpenAI Ada). We then store these vectors along with metadata like chunk ID, document ID, chunk text, etc.

Vector Database: We have options for storing embeddings: - **PostgreSQL with pgvector:** This is a Postgres extension that allows a column type `VECTOR` for embeddings and provides an `<->` operator for similarity search (cosine or L2). It's simple and keeps everything in one database [5]. We can create a table `contract_chunks(chunk_id SERIAL, contract_id INT, text TEXT, embedding VECTOR(1536))` and create an index on the embedding. - **OpenSearch:** OpenSearch (the open-source fork of ElasticSearch) supports dense vector fields and KNN search, which can combine with traditional keyword filters (hybrid search). This could be useful for precise retrieval (e.g., ensure the chunk contains certain keywords AND is semantically similar). However, operating an OpenSearch cluster is heavier than Postgres. For a small team MVP, pgvector is often sufficient, or even an embedded vector store like **chromadb** or **FAISS** in memory. - **Managed Vector DB:** If we prefer not to self-manage, services like Weaviate or Pinecone can host vectors (the Blackletter plan mentions free tiers for these). They offer hybrid search as well. The tradeoff is complexity and potential data leaving our environment. Given privacy concerns, using our own Postgres (especially if we already need it for other data) is a straightforward choice.

Storing Embeddings: With Postgres pgvector, after enabling the extension (`CREATE EXTENSION vector;`), we can do something like:

```sql
CREATE TABLE contract_chunk (
   id SERIAL PRIMARY KEY,
   contract_id INT,
   content TEXT,
   embedding VECTOR(1536)  -- length depends on model
);
-- Also store an index for faster search (approximate or exact)
CREATE INDEX idx_contract_chunk_embedding ON contract_chunk USING ivfflat
(embedding vector_cosine_ops);
```

In Python, using e.g. `psycopg2` or SQLAlchemy, we would insert each chunk's data and the embedding (as a list of floats). When we need to query, we compute the embedding of the query or question and use a SQL query:

```sql
SELECT content, embedding <-> query_embedding AS score
FROM contract_chunk
WHERE contract_id = 123
ORDER BY query_embedding <-> embedding
LIMIT 5;
```

This returns the top 5 most similar chunks. (The `<->` operator computes distance/similarity [8].)

Hybrid Search: To ensure we don't miss obvious keyword hits, we can combine semantic similarity with keyword filtering. For example, if checking for a "breach notification" clause, it makes sense to require the word "breach" or "72 hours" be present. In a vector search, if the contract completely lacks that concept, the top vector match might be irrelevant. One strategy: if rule engine found a keyword hit, directly use that chunk; if not, still do a vector search with query "data breach notification" to see if something semantically similar pops (in case wording is different, e.g., "security incident"). Alternatively, do a preliminary keyword search (even a simple substring search in text) – if nothing, we know it's likely missing. We can instruct the LLM accordingly (more on that in LLM section).

Indexing Pipeline: The background worker after parsing could perform: 1. `chunks = split_text_to_chunks(text)` 2. For each chunk: `vec = embed(chunk)` using chosen model. 3. Insert (contract_id, chunk_text, vec) into the vector store.

For speed, embedding API calls can be batched (OpenAI allows batches of up to 16 or more). If using a local model, we can also batch process.

Privacy Consideration: Storing chunks and embeddings means sensitive content is in the DB. Ensure the database is access-controlled and encrypted. Since we only retrieve by similarity, not by exact matching, someone with DB access would still see chunk text. Therefore, all DB access should be through our app (no direct user access) and credentials kept secret (Vault). If extreme privacy is needed, chunks could be transient (not stored after processing) – but then we couldn't regenerate reports without re-processing. A balanced approach is to keep the chunks for a limited time (e.g., auto-delete them after X days along with the rest of data, as per retention policy).

Scaling: For MVP scale (few documents, each maybe a few pages), Postgres with a couple hundred embeddings is trivial. If we had thousands of large documents, we'd consider performance tuning (using approximate search indices like HNSW in pgvector or moving to a dedicated vector DB). But within the given budget, this setup will handle dozens of documents per day easily on a small VM.

LLM Judge – Compliance QA with Citations

The **LLM Judge** is a core intelligent component: it uses a Large Language Model to assess the content of the contract against GDPR requirements. Unlike static rules, the LLM can understand context and interpret whether the contract's language truly satisfies a requirement or not. Crucially, we constrain the LLM with retrieved context (from the RAG step) and require it to provide **citation of sources** for its conclusions.

Prompt Structure: We use a carefully designed prompt to query the LLM. This usually involves a *system message* setting the role and format, and a *user message* with the question and context. For example, using OpenAI's Chat API:

- **System message:** "You are a legal AI assistant specialized in GDPR. You help determine if a contract meets certain GDPR clauses. You will be given an excerpt of a contract and a specific requirement. If the requirement is met, answer YES with an explanation and quote the relevant contract text. If not, answer NO with an explanation and, if possible, cite where it falls short. Always provide a reference to the contract excerpt in brackets."
- **User message:** Could be something like: *"Requirement: The contract must require the processor to notify the controller of personal data breaches within 72 hours. Contract excerpt: '<chunk text>'. Question: Does this contract excerpt meet the requirement?"*

The contract excerpt is the chunk (or combination of chunks) retrieved by the RAG search for that topic. By isolating the excerpt, we ensure the LLM only sees a small portion of the contract (privacy) and stays focused. If the rule engine found nothing (i.e. likely missing clause), we might not have an excerpt. In that case, we can prompt the LLM with something like: *"The contract text does not appear to contain any clause about breach notification within 72 hours. Question: Is the contract compliant with the breach notification requirement?"* The LLM should answer NO and perhaps add that no such clause was found.

Ensuring Citations: We want the LLM's output to cite sources (the contract text it used). We enforce this via prompt instructions and possibly output format constraints. A strategy is to provide an example in the system prompt: e.g., *"Format: 'YES/NO – Explanation... (citing "...quote..." from contract).'"* Another approach shown by Anthropic's Claude and research is using special tags [3]. For instance, we could instruct the model to output citations as `<CIT>` tags with an identifier for the chunk [3]. Since we know which chunk we provided, even a simple `[Source]` or `[Clause 5]` in the answer could be enough, and we can post-process that. The Medium article on Anthropic-style citations suggests embedding markers around the specific sentences used [9], which can then be mapped back. For MVP, a simpler method: ask the model to explicitly quote the relevant text in the answer.

Example LLM Q&A:

User prompt:
"**Requirement:** The contract must include an end-of-contract provision requiring the deletion or return of personal data upon termination (GDPR Art. 28(3)(g)).
Contract Excerpt: *'Upon termination of the Agreement, the Processor shall, at the choice of the Controller, delete or return all personal data and delete existing copies unless applicable law requires retention of the personal data.'*
Question: Does the excerpt satisfy the requirement?"

Assistant (ideal answer):
"Yes. The contract excerpt explicitly states that *'upon termination... [the Processor shall] delete or return all personal data...'*, which **meets the GDPR requirement** for end-of-contract data handling. The clause ensures personal data is either returned or deleted when the contract ends, as required by GDPR Article 28(3)(g)."

Here the assistant quoted the source text as evidence. In a more complex scenario, it might cite a specific clause number or page – we can supply that info in the prompt if available (e.g., we could tell the model "this text is from Clause 12 of the contract" and then it can say *"Clause 12"* in the answer). For now, quoting the text or referring to it implicitly is acceptable. The key is the output contains something that clearly derives from the provided snippet, not a hallucinated reference.

Handling Non-compliance: If the excerpt provided does not meet the requirement (or if no excerpt was found), the LLM should say “No” and explain. For example: *“No, the contract does not explicitly require breach notification within 72 hours. I did not find any clause mentioning breach notification or a 72-hour timeframe, so this requirement appears to be missing.”* This explanation again references the lack of text as evidence.

Model Choice: Given the complexity of legal text, **GPT-4** would be ideal for the judge due to its better reasoning. However, GPT-3.5 (turbo) can be used to save cost and still handle straightforward checks. Another option is **Anthropic Claude** which has the citation feature (but its availability/cost might vary). Open-source LLMs (like LLaMA 2 13B with fine-tuning) could potentially be used on-prem to avoid external API, but likely require a GPU and might not be as reliable out-of-the-box for legal nuance. For MVP, using an API like OpenAI is faster to implement. We ensure not to send more data than needed and possibly use their data-protection settings (OpenAI allows opting out of data retention on their side).

Prompt Implementation: The prompt can be assembled in `llm_judge.py`. We might use a library like **langchain** or just direct API calls. Pseudo-code:

```
import openai
def analyze_clause(requirement: str, snippet: str):
    system_msg = {"role": "system", "content": (
       "You are an expert legal assistant for GDPR compliance. "
       "Answer the question about the contract excerpt truthfully. Cite the
contract text as evidence in your answer."
    )}
    user_msg = {"role": "user", "content": (
       f"Requirement: {requirement}\n"
       f"Contract Excerpt: \"{snippet}\"\n"
       "Question: Does this contract excerpt fulfill the requirement?"
    )}
    response = openai.ChatCompletion.create(model="gpt-4",
messages=[system_msg, user_msg])
    answer = response['choices'][0]['message']['content']
    return answer
```

The returned `answer` might include a Yes/No and explanation. We then parse it (if needed) or directly include it in the report. We should double-check the answer for citations. If we demanded a certain format (like starting with "Yes" or "No"), we could verify the output string and possibly post-process (e.g., ensure it contains at least one quote from the snippet).

To further **enforce format**, we could use few-shot examples in the prompt. For instance: *“Example - Requirement: X, Excerpt: "...". Q: ... A: Yes. Because "...quote...".* Providing one or two examples can guide the model to always answer with the pattern. Another advanced technique is two-pass: have the model first list relevant snippet sentences then answer, but that’s likely overkill here [10]. We prefer a single-pass prompt for simplicity.

Citation in Output: The output might contain quoted text or references like `[CIT]`. For the final report, we can highlight those. If we had chunk IDs, we could map “[CIT1]” to something like a footnote with the actual clause text. For now, since the model is directly quoting, we can just present that text in quotes or italics in the report. The important part is that for every claim of compliance or non-compliance, the model points to the contract language (or explicitly notes the absence of it) [3] . This gives the user confidence and an audit trail.

Rate Limiting & Cost: We must account for API rate limits and costs. If we have, say, 10 rules to check via LLM, that's 10 API calls per document. If using GPT-4, that could be ~$0.03 per call (depending on prompt length, maybe 1-2K tokens round-trip), so 10 calls = $0.30 per document – well within budget if not doing thousands. GPT-3.5 would be an order of magnitude cheaper. We can also optimize by only calling LLM on uncertain cases: e.g., if rule engine found a clause, ask LLM to verify quality; if rule engine found none for a required clause, one could argue no need to call LLM to confirm it's missing, though it might be nice to have the LLM phrase the issue. To save cost, we might skip LLM for the obvious misses and just use a template explanation for those. Alternatively, one combined prompt could ask the model to analyze all requirements in one go with the relevant excerpt for each – but that becomes a very large prompt and harder to parse. The per-item approach is simpler and parallelizable.

Citation Integrity: As a final check, since we are automating compliance, we might want to verify that any text the LLM "quoted" actually appears in the contract. A simple mechanism: for each answer, extract the text inside quotes or any chunk of 5-10 words, and search the original contract text to ensure it's there. This guards against hallucinations. If a quote isn't found, we could mark that answer as unverified or re-prompt the LLM. This kind of check adds reliability given LLMs sometimes make up official-sounding text. Because we constrain the context to the snippet we fed, the chance of hallucination is low (it can't fabricate too much if all it has is the snippet). Nonetheless, a programmatic sanity check using string search can be implemented as a safety net.

Report Generation – HTML & PDF Outputs

After rule checks and LLM analyses are done, the system compiles the results into a **compliance report**. This report gives a summary of which GDPR requirements are satisfied and which are not, along with explanations and remediation recommendations. It can be presented on the web app and also offered as a downloadable PDF (and possibly other formats like a Word document with redlines).

Report Content: A typical report structure might be: - **Title**: e.g., *"GDPR Vendor Contract Compliance Report"* with date, maybe contract name. - **Summary:** a brief overview stating something like "Out of 8 key GDPR clauses, 5 are present and adequate, 3 are missing or insufficient." It could include a quick risk level (e.g., high/medium based on which clauses are missing). - **Details by Clause:** A section for each rule (or category of rules): - Requirement description (e.g., "Breach Notification: Vendor must notify data breaches within 72 hours"). - Compliance status: ✔ **Yes** (compliant) or ✖ **No** (not compliant) or maybe a warning icon for partial compliance. - Explanation: If compliant, a sentence like "The contract contains a clause requiring breach notification within 72 hours [7] ." If not, "No clause found addressing breach notifications; this is required by GDPR – the contract should include a term obligating the processor to inform the controller of any data breach within 72 hours." - Citation: If a clause was found, quote the relevant snippet (as evidence). If none, perhaps cite GDPR or best practice: e.g., *"(GDPR Art 33 requires breach reporting within 72h)."* We might link out to sources or just mention them. - **Vague Terms Warnings:** A list of any flagged vague terms with suggestions to clarify them. For example, "The contract uses the term 'appropriate measures' – consider specifying exact measures or standards (ambiguity can lead to disputes)." - **Recommendations:** A final section listing what changes to make to achieve compliance. This can be directly drawn from the `if_missing` texts of rules (e.g., "Add a clause requiring X"). If we have some boilerplate clauses or references, include them. For instance, "Include a data return or deletion clause. E.g.: 'Upon termination, the processor will return or delete all personal data…'". - **Appendices (optional):** Could include the full text of the contract with highlights, or a glossary. In MVP, we probably skip this, but we might provide a link to download an annotated contract.

HTML Generation: We can create an HTML template for the report. Using a template engine like **Jinja2** (for Python) is convenient. We can fill in dynamic content into an HTML skeleton. For example, `report_builder.py` might have:

```
from jinja2 import Template

REPORT_TEMPLATE = Template("""
<html>
<head><style>
  body { font-family: Arial, sans-serif; }
  .status-pass { color: green; }
  .status-fail { color: red; }
  /* ... more styles ... */
</style></head>
<body>
  <h1>GDPR Contract Compliance Report</h1>
  <p><strong>Contract:</strong> {{ contract_name }}<br>
     <strong>Date:</strong> {{ date }}</p>
  <h2>Summary</h2>
  <p>{{ summary }}</p>
  <h2>Clause-by-Clause Analysis</h2>
  {% for issue in issues %}
    <h3>{{ loop.index }}. {{ issue.description }}</h3>
    <p>Status:
      {% if issue.compliant %}<span class="status-pass">Compliant ✔</span>
      {% else %}<span class="status-fail">Not Compliant ✖</span>{% endif %}
    </p>
    <p>{{ issue.details }}</p>
    {% if issue.citation %}
      <blockquote>{{ issue.citation }}</blockquote>
    {% endif %}
  {% endfor %}
  <h2>Recommendations</h2>
  <ul>
    {% for rec in remediation %}
      <li>{{ rec }}</li>
    {% endfor %}
  </ul>
</body>
</html>
""")
```

This is a simplified template. We would render it by passing the context: `REPORT_TEMPLATE.render(contract_name=name, date=today, summary=summary_text, issues=issues, remediation=recommendations)`.

The `issues` list would contain each rule's outcome (with description, compliant bool, details text, citation text). The `remediation` list could be compiled as all `if_missing` messages of failed rules, or some more action-oriented phrasing. Possibly, some remediation items might be static suggestions from the rules file.

PDF Generation: To produce a PDF, a straightforward way is to convert the HTML. Tools like **WeasyPrint** (a Python library) or **wkhtmltopdf** can take HTML/CSS and output PDF. WeasyPrint is pure Python and was mentioned in the plan for generating PDFs [11] . We could do:

```
from weasyprint import HTML
def html_to_pdf(html_content, output_path):
    HTML(string=html_content).write_pdf(output_path)
```

This requires that the CSS in the HTML is reasonably simple (WeasyPrint supports a good subset of CSS). Our inline styles or a linked CSS should work as shown. Alternatively, Next.js could handle report rendering on the client side and a user can print to PDF, but automating it on the backend ensures consistency (and allows us to zip up a package of files to send via email, for instance).

Redlined Contract (Stretch Goal): The plan hints at generating a redlined DOCX [4] – basically inserting suggestions or track changes into the original contract. This is a nice feature for lawyers: instead of just telling them what's wrong, provide a marked-up contract with corrections. Implementing that is non-trivial but possible: - Using `python-docx`, one can add comments to a Word doc at specific points. We could locate where a clause should be added or changed and insert a comment like “Add breach notification clause here.” However, accurate placement is hard without a structural parse of the contract. - Alternatively, create a new Word document that contains all the suggested clauses, as an addendum or as an edited version of original text. - True track changes (Word’s redline) involves Word-specific formats (not directly supported by python-docx as far as adding revision marks). For MVP, a simpler path is to list suggestions in the report, and allow the user to download the suggestions as a Markdown or Word document (not necessarily merged into the original). Given time constraints, we may skip generating a redlined DOCX and focus on the report and summary. We do, however, log all needed info so a human can manually edit the contract.

Packaging Output: The results can be stored in the database (the JSON of issues) and the PDF stored in the file storage. The API can then provide a URL or endpoint to download the PDF (`report_url` as in the schema). Possibly, we also keep the HTML or Markdown summary for display in the app (or we could regenerate it on the fly from the DB data when the user opens the report page – which might be easier than storing HTML).

Example Output (HTML snippet):

```
<h3>1. Breach Notification (72-hour notice)</h3>
<p>Status: <span style="color:red;">Not Compliant ✖</span></p>
<p>No clause was found requiring the processor to notify the controller of
personal data breaches within 72 hours. This is required under GDPR Article
33; the contract should be amended to include a breach notification
obligation.</p>

<h3>2. Data Deletion on Termination</h3>
<p>Status: <span style="color:green;">Compliant ✔</span></p>
<p>The contract includes an end-of-contract provision for data return or
deletion.
   <blockquote>“...the Processor shall, at the choice of the Controller,
delete or return all personal data... upon termination of the Agreement.”</
blockquote>
```

```
    This meets the GDPR requirement for post-termination data handling.
</p>
```

This illustrates how we show a failing clause vs a passing clause with evidence. Notice the failing one has no excerpt, just an explanation and recommendation. The passing one quotes the text.

We also ensure the report emphasizes privacy and next steps: e.g., *"All analysis data will be deleted after 30 days by default."* This can be a footer note to reassure users (privacy-by-design).

Asynchronous Workflows – Celery & Background Tasks

Processing a contract involves CPU-intensive work (OCR, embedding) and potentially slow API calls (LLM), so we decouple it from the user request using asynchronous background workflows. This improves responsiveness and reliability [12]. There are two primary ways to handle this: **Celery** (a distributed task queue for Python) or **Temporal** (a workflow orchestration system with multi-language support). We'll focus on Celery for simplicity.

Celery Setup: Celery consists of: - A **broker** (e.g., Redis or RabbitMQ) to which producers (the API) enqueue tasks. - One or more **worker** processes that listen on queues, execute tasks, and report results. - An optional **result backend** (could be Redis, DB, or just using the DB we have) to store task outcomes.

In our `workers.py`, we configure Celery:

```
from celery import Celery

celery_app = Celery('blackletter', broker=os.getenv('CELERY_BROKER_URL',
'redis://localhost:6379/0'))
# (optional) use database as result backend
celery_app.conf.result_backend = os.getenv('CELERY_RESULT_BACKEND', 'redis://
localhost:6379/0')
```

We can organize tasks into separate modules or define them inline. For example, a single workflow task:

```
@celery_app.task(name="process_contract")
def process_contract(job_id: int):
    # 1. Load job and file path from DB
    job = db.get_job(job_id)
    text = extract_text(job.file_path)         # from ingest.py
    text = perform_ocr_if_needed(text, job.file_path)
    db.save_contract_text(job_id, text)
    # 2. Run rule engine
    rule_issues = run_rules_on_text(text)      # from rules_engine.py
    db.save_rule_results(job_id, rule_issues)
    # 3. Prepare RAG vectors
    chunks = split_text_to_chunks(text)
    embeddings = embed_chunks(chunks)
    store_embeddings(job_id, chunks, embeddings)
    # 4. LLM analysis for each rule that requires verification
    llm_issues = []
    for issue in rule_issues:
        if issue.rule_id in ["vague_terms"]:  # maybe skip LLM for vague
terms or missing clauses
            continue
        if issue.compliant is True:
            # clause exists, verify quality
            snippet = find_relevant_chunk(issue.rule_id, chunks)  # pseudo:
pick chunk by rule or regex
            question = issue.description  # using description as requirement
question
            answer = analyze_clause(issue.description, snippet)   # call LLM
judge
            llm_issues.append(parse_llm_answer(issue.rule_id, answer))

# If issue.compliant is False and required (clause missing), we could ask LLM
for impact or just mark it
    db.save_llm_results(job_id, llm_issues)
    # 5. Compile report
    report = build_report(job_id, rule_issues, llm_issues)
    pdf_path = generate_pdf(report.html)
    db.update_job(job_id, status="completed", report_path=pdf_path)
```

This pseudo-code illustrates a linear pipeline inside one Celery task. We could break these into multiple tasks for clarity or parallelism: - `parse_document` task, - `embed_document` task (depends on parse), - `rule_check` task, - `llm_analysis` task, - `build_report` task.

Celery allows making task workflows via **chains** or chords. For example, one could do:

```
chain(parse_document.s(job_id) | analyze_document.s() | build_report.s())()
```

or use Celery Canvas to orchestrate. However, for an MVP, it might be overkill to split every step unless we want parallel execution (like multiple LLM calls concurrently). Given that LLM calls could be the slowest, we might indeed dispatch them in parallel using a group chord: e.g., group tasks for each rule's LLM check and then callback to compile results.

If using **Temporal** (or similar), we'd define a workflow function that calls activities for each step, with Temporal handling scheduling and retries. Temporal has the advantage of stateful workflows and better failure recovery (Celery tasks are more fire-and-forget with manual retry logic). But Temporal's setup is heavier (requires running a Temporal server and using its SDK). For a small team and limited budget, Celery (or even simpler, Python `concurrent.futures` threads if tasks are CPU-bound) is sufficient. The Reddit RAG example used Celery to offload document processing to keep the API responsive [12], which is exactly our use case.

Job Monitoring: With Celery, when we queue a task, we can get a task ID. The API can return that to the client. We can then have an endpoint `/status/{job_id}` which checks the database or Celery result backend for completion. In the above flow, we updated the DB status at end; we could also poll Celery's AsyncResult. Many use a simple approach: have a job table with status field (queued, processing, done, error) that the worker updates as it goes. This is straightforward and also allows showing progress (we could update status like "OCR complete", "Embedding complete", etc., but that might be too granular for MVP).

Parallel LLM Calls: One optimization: after embedding, for each rule that needs LLM verification, we could spawn a subtask (Celery group) to call `analyze_clause`. Celery can run them concurrently if we have multiple worker processes or threads. Once all are done, a chord callback compiles the results. This would speed up overall processing when multiple LLM checks are needed, at the cost of using more concurrency (still likely within API rate limits unless we spawn too many at once).

Resource Management: We must be mindful of memory and CPU: - The OCR (Tesseract) and embeddings can be CPU heavy. If running on a single machine, the Celery worker can be configured to use multiple processes or threads. Alternatively, separate queues can be designated: e.g., one queue for CPU-bound tasks (with concurrency=1 or 2 if OCR uses lots of CPU) and another for I/O or quick tasks. - LLM API calls are I/O bound (network waits), so those could be done concurrently without heavy CPU. - We can configure Celery to use a concurrency level that matches machine cores. Also, since we might have at most a handful of jobs at a time, a single worker process could sequentially handle them fine.

Temporal Alternative: If we had more time/infrastructure, we could use Temporal (with Python client or Node). Temporal would let us define each step as an activity and the whole sequence as a durable workflow. That ensures even if the worker crashes mid-way, it can continue from where it left off. Celery can also retry tasks, but if a task halfway fails, you'd have to handle partial progress manually. For MVP, Celery's retry (e.g., you can decorate tasks with `autoretry_for=(Exception,)` to auto-retry on failure, or manually catch and retry) is usually enough. We should implement some error handling: if an LLM call fails or times out, catch and perhaps retry once, or mark that particular analysis as unable to complete (but still produce a report, noting an error for that clause).

Example Celery usage (from Ragpi): *"Celery is used to handle time-consuming tasks, specifically processing documents (fetching, chunking, embedding, storing)… offloading these tasks to Celery workers allows the API to remain responsive* [13] *."* Our system mirrors this approach exactly. We also instrument tasks with logging or OpenTelemetry spans for observability (see next section).

Data Storage & Infrastructure – PostgreSQL, S3 & Vector DB

The persistence layer ensures that we can reliably store and retrieve data across the workflow and between sessions. We utilize **PostgreSQL** as the primary database, an **S3-compatible storage** for files, and integrate vector capability either within Postgres (pgvector) or via an external service.

PostgreSQL (Relational DB): Postgres will store: - **Job metadata:** job ID, user/tenant, file name, status, timestamps, etc. - **Extracted text (optional):** possibly in a `contract_text` table (job_id, text) if we want to preserve it. - **Rule check results:** could be stored as a JSON or separate table linking job->issues. For example, an `issue` table with columns (job_id, rule_id, compliant, details, citation). - **Embeddings:** if using pgvector, a `contract_chunk` table as described earlier holds embeddings and text for each chunk. - **Users/tenants:** if multi-tenant, tables for user accounts, perhaps API keys, etc. - **Playbooks/rules:** although we have YAML, we could also store them in a table or at least store which playbook was used for a job. - **Reports:** maybe store the final HTML or PDF path for quick retrieval.

Using Postgres has the advantage of consistency (all data in one place, easy to backup) and cost-effectiveness (one small managed DB instance or container can handle all needs). The pgvector extension brings vector search into the same DB; installing it is usually one command. If using a cloud Postgres like Supabase or Timescale (as mentioned in the plan), they often have pgvector available or an easy way to add it [5] .

Vector Storage Alternatives: If we decide not to use pgvector, the alternative is **OpenSearch** or a specialized vector store: - **OpenSearch:** We could run a single-node OpenSearch (which uses ~2GB RAM minimum) to store both full-text and vectors. We'd push each chunk as a document with fields for text, maybe rule tags, and a vector. Then query using its API. This adds complexity in deployment and maintenance – likely not necessary for MVP scale. - **Redis (Redis Stack):** Redis now supports vector similarity (in Redis Stack). One could use Redis both as Celery broker and to store vectors in a Redis index. This is an interesting approach (the Ragpi project used Redis as vector DB) [14] . It keeps latency low, but persistence might be an issue (unless using Redis persistence or snapshotting). It could work if we want a quick in-memory store for embeddings with minimal setup. However, since we need Postgres anyway for other data, having two datastores might be unnecessary. - **Chroma or Qdrant:** These are lightweight vector DBs you can run as a container, but again it's another component to manage.

For our budget-conscious design, **embedding everything in Postgres** is appealing for simplicity and cost (just one service to host). Performance for a few hundred vectors is fine; for thousands, pgvector with an approximate index is still okay. Only if we expected millions of vectors or very high QPS would a dedicated vector DB be needed.

Object Storage (S3/MinIO): All uploaded documents and generated reports are stored in an S3-compatible bucket. During processing, we might use a local copy, but ultimately we want them persisted beyond the ephemeral worker. For example: - When a file is uploaded, the API could directly stream it to S3 (using AWS SDK or presigned URL via the browser). Alternatively, the API saves it locally then later moves to S3. But saving directly to S3 is more scalable (especially if the API is serverless on something like Vercel, which can't hold files). - We could use AWS S3, or since budget is a concern, something like **Backblaze B2**, **Wasabi**, or even **Supabase Storage** which wraps S3, could be used (Supabase's free tier might cover some storage). - For local development, **MinIO** (which emulates S3) can run as a Docker container. We might include that in docker-compose for testing.

We should define an S3 bucket name, and folder structure, e.g.:

```
s3://myapp-bucket/uploads/{tenant_id}/{job_id}/original.pdf
s3://myapp-bucket/reports/{tenant_id}/{job_id}/report.pdf
```

Storing per tenant in separate prefixes aids deletion (you can delete a whole prefix if needed) and isolation.

Deletion & Retention: Implement a routine (could be a daily cron job or simply on-demand) to delete objects and DB records older than the retention period (e.g. 30 days) [6] . This ensures we don't accumulate sensitive data. Also provide an API endpoint to delete a specific job on user request (this would remove the DB entries, vectors, and S3 files for that job).

Encryption: We enable encryption at rest: - S3: Enable server-side encryption (AES-256 or KMS-managed) on the bucket so all files are encrypted. If using MinIO, we can configure it similarly. - Postgres: If using a managed service, typically the disk is encrypted. If self-managed, ensure the volume or RDS instance has encryption turned on. Additionally, use TLS for all connections (HTTPS for S3, TLS for Postgres) especially if services are cloud-hosted.

Vault for Secrets: We store sensitive config like DB connection URL, S3 keys, OpenAI API key in HashiCorp Vault, or at least in environment variables that are not checked into code. If using Vault, one way is to have the app on startup query Vault (which requires Vault address and token). For MVP, an .env file or using something like Doppler or a cloud secret manager might be simpler. The mention of Vault is to highlight enterprise-grade secret management – since this is a compliance-oriented tool, using Vault shows commitment to security. One could run Vault in dev mode for local testing (or use environment variables as a fallback when Vault is not configured).

Vault usage example: Store secrets (like `openai_api_key`, `database_url`) in Vault's KV store. At container startup, the entrypoint script uses Vault CLI or API to fetch them and set environment variables for the app. Alternatively, integrate Vault agent to auto-inject secrets into a file. This can be part of Terraform provisioning (Terraform can populate Vault or ensure the app has access).

Tenant Isolation: We enforce that each tenant's data is separate. In the DB, every table that stores user data has a `tenant_id` or `user_id` column. All queries in the backend filter by that based on the authenticated user's context. E.g., when a user requests `/review/{job_id}`, the backend will ensure the job's tenant_id matches the requesting user's tenant, otherwise return 404 or forbidden. This prevents any chance of one customer seeing another's contract. On the vector side, we can partition by tenant by including tenant_id in the contract_chunk table and adding it as a condition in similarity queries (e.g., `... WHERE tenant_id = X ORDER BY embedding <-> query_embedding ...`). Alternatively, one could have separate schema or even separate databases per tenant, but that's heavy for MVP. A simple row-level isolation is fine if carefully implemented.

File access: Similarly, if using S3, ensure either separate buckets or at least object keys are unpredictable (including job UUIDs) and don't expose list permissions across tenants. Usually, the backend will generate signed URLs for download only to authorized users.

Scaling and Cost: With the given budget: - We can use a small Postgres instance (for example, a $15/month digital ocean DB or AWS RDS on free tier initially). - S3 storage cost will be minimal (a few MB per contract). - Redis for Celery could be a small managed instance or even use Redis in Docker if on one server. - Running everything on a single VM vs separate: Possibly we could run Postgres and Redis on the same VM for very small scale, but using managed services often is worth it for reliability. - If using Supabase: it provides Postgres and storage free up to certain limits, which might actually cover initial usage at zero cost.

We should plan infra such that it can run on one machine for dev, but scale to separate components in cloud. Using Docker containers for each piece ensures portability.

Next, we discuss observability and deployment.

Observability & Security – OpenTelemetry, Vault, and Privacy-by-Design

Building user trust in this system is paramount, as it will handle sensitive legal documents. We achieve this through robust **observability** (so we can audit and debug the system's behavior) and stringent **security and privacy practices** at every layer.

OpenTelemetry (OTel): We instrument the application with OpenTelemetry for distributed tracing and metrics [15]. This means: - Adding OTel middleware to FastAPI or NestJS to trace incoming HTTP requests. There are libraries like `fastapi-instrumentator` or the OTel SDK that can automatically trace requests, including request/response info (excluding sensitive payloads). - Instrumenting Celery tasks: The OpenTelemetry Python instrumentation for Celery can propagate trace context from the API to the worker, allowing end-to-end trace of a job (from HTTP request, through enqueue, to each task). - Instrumenting database calls (OTel has an integration for psycopg2/SQLAlchemy) and external HTTP calls (like to OpenAI) to record their latency. - These traces can be exported to a tracing backend. For local/dev, we could use **Jaeger** (an open source distributed tracing UI). In production, a SaaS like DataDog, Honeycomb, or Grafana could be used if budget permits (some have free tiers). - Additionally, define some custom **metrics**: e.g., counter of how many contracts processed, histogram of processing time, etc. These can be pushed to a Prometheus or StatsD. This is optional but useful for monitoring usage and performance.

By having tracing, if a job is stuck or fails, we can see which step took long or where an exception occurred. Also, we can prove that data flows are as expected (for auditing) – e.g., that the system called out to only authorized endpoints.

Logging: We implement structured logging in both API and workers. All logs should exclude sensitive data. For instance, do not log the content of the contract text or any PII. Instead, log high-level events: "Job 123: file uploaded by TenantA", "Job 123: parsed text length 10,000 chars", "Job 123: missing breach clause", etc. These logs can help in debugging issues and also provide an audit trail. If required by the user, we could expose an "audit log" of actions (like who uploaded what when, etc.). For now, logs mainly serve developers.

Security Measures: - **Secret Management:** As discussed, use environment variables or Vault for secrets. This prevents secrets from leaking in code repositories. Limit access – e.g., if using Vault, it can also dynamically provide short-lived DB credentials (advanced use case, perhaps not needed initially). At minimum, one would have a `.env` file (never committed to Git) with `DATABASE_URL`, `OPENAI_API_KEY`, etc., and the production deployment injects those securely. - **Least Privilege:** Ensure that each component only has the access it needs: - The database user should have access only to its schema. Use a separate user for application vs admin. - The S3 credentials should only allow access to the specific bucket (and possibly even prefix) used by the app, not all buckets. - If running on AWS, consider IAM roles for the app that only allow those actions. - Within the app, if multi-tenant, ensure tenants cannot act as admin or see each other's data as discussed. - **File Scanning:** The plan mentions file scanning [6]. This implies scanning for viruses or malware in uploaded files. It's a good practice since users might upload any PDF/DOCX. We can integrate ClamAV (open-source antivirus) or an API like VirusTotal for this. ClamAV can be run in a container; after upload, before parsing, run a scan on the file. If malware is detected, reject the file and alert. This prevents a malicious file from exploiting our parser or from being passed on to an analyst later. - **Access Control:** The frontend should have an authentication system (even if simple). For an MVP, maybe a simple email/password or magic link (services like Supabase Auth or Clerk can be set up quickly). This ensures only authorized users (and their team) can upload/view their contracts. All API routes should be protected, i.e., require a valid token or session. FastAPI can use OAuth2 with JWTs, NestJS has guards, etc. This is more about product usage security than internal, but it's important if multi-tenant.

Privacy-by-Design Defaults: We implement the system so that privacy is not an afterthought: - **Minimal Data to LLM:** We only send necessary snippets (not entire documents) to the LLM for analysis. In addition, if feasible, we anonymize those snippets (e.g., replace company names with generic tokens). This greatly limits exposure of sensitive info. For example, instead of sending "ACME Corp shall notify Example Inc. of breaches within 72 hours", we might send "[Vendor] shall notify [Client] of breaches…" which preserves the clause meaning but anonymizes the parties. - **Data Retention:** By default, any uploaded contract and its analysis are scheduled for deletion after, say, 30 days [6]. We inform the user of this in the UI and maybe allow them to expedite deletion or extend it if needed. A background job or a simple daily script can query the DB for any jobs older than 30 days and: - Delete the file from S3, - Delete the DB records (or at least anonymize them if we need to keep statistics). - Delete vector entries. Essentially a thorough scrub. This compliance with storage limitation is important for GDPR as well (since ironically our tool must also follow GDPR). - **User-initiated Deletion:** Provide an option in the UI to “Delete this report/contract” which triggers immediate deletion of that job’s data as above. This respects user’s right to erasure. - **Encryption:** We already covered encryption at rest. Also enforce HTTPS for any network communication (frontend to API, API to external services). If self-hosting, get TLS certificates (maybe via Let’s Encrypt). - **Audit Logging:** Keep an audit log (even if just in a log file or DB table) of who accessed or deleted data. This is useful if any security incident or user question arises: we can show records of data access.

Testing Security: Before deployment, we should test with some sample documents containing known PII (to see if we properly redact), test the multi-tenant isolation (create two dummy accounts, ensure one cannot fetch the other’s data by manipulating IDs), and test deletion (verify no trace in DB or storage after deletion).

Compliance of the Tool: Since this is a GDPR compliance tool, it would be prudent to also ensure the tool itself doesn’t violate GDPR. For instance, if using OpenAI API, we should ideally inform users that their contract text is being sent to a third-party AI service. OpenAI allows opting out of data usage for training, which we should do by sending the `OpenAI-Organization` and opting-out flags (OpenAI’s policy as of 2025 is not to use API data for training if opted out or on paid plans). Alternatively, if a client insists on no external data transfer, we’d need to use an on-prem LLM. This could be addressed on a case-by-case basis.

Vault Integration: Assuming we have Vault set up, our application needs to authenticate to Vault (usually via a token or an AWS/GCP auth method). In a simple scenario, we might not actually integrate Vault in code but use it during deployment: e.g., use Terraform to pull secrets from Vault and supply them as env vars to the containers. If we do integrate at runtime, we’d use the Vault API to read a key-value path:

```python
import hvac  # HashiCorp Vault API client
vault_client = hvac.Client(url=VAULT_ADDR, token=VAULT_TOKEN)
secrets = vault_client.secrets.kv.v2.read_secret_version(path='blackletter/
app')
openai_key = secrets['data']['data']['OPENAI_API_KEY']
```

We would do this at startup and set the configurations accordingly. The advantage is we never store the actual key on disk, and rotating it in Vault means just restarting the app loads the new key.

In summary, the system is built with security and privacy from the ground up: *least privilege, encrypted data, minimal exposure, and user control over their data.* This not only protects us and the users, but it also aligns with GDPR principles, which is important since we are positioning this as a GDPR tool (it should practice what it preaches).

Deployment – Local Development and Cloud Setup

We design deployment to be as smooth as possible, using **Docker** for containerization and **Infrastructure-as-Code** for reproducibility. The goal is that a small dev team can run everything locally, and then deploy to cloud with minimal changes, staying within the startup and monthly cost budget.

Local Development: The repository includes a `docker-compose.yml` that defines services for each component: - `frontend` (Next.js dev server or build), - `backend` (FastAPI Uvicorn server), - `worker` (Celery worker), - `db` (Postgres), - `redis` (for Celery broker), - `minio` (for S3). - Optionally `vault` (in dev mode) and maybe `jaeger` for tracing.

A simplified docker-compose might look like:

```
version: '3.8'
services:
  db:
    image: postgres:15
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
      - POSTGRES_DB=blackletter
    ports: ["5432:5432"]
  redis:
    image: redis:7
    ports: ["6379:6379"]
  backend:
    build: ./backend
    env_file: .env             # contains DB URL, OpenAI key, etc.
    depends_on: [db, redis]
    ports: ["8000:8000"]
  worker:
    build: ./backend           # using same image as backend, but override
entrypoint
    command: celery -A workers.celery_app worker --concurrency=1
    env_file: .env
    depends_on: [db, redis]
  frontend:
    build: ./frontend
    environment:
      - NEXT_PUBLIC_API_URL=http://localhost:8000
    ports: ["3000:3000"]
  minio:
    image: minio/minio
    environment:
      - MINIO_ROOT_USER=minio
      - MINIO_ROOT_PASSWORD=miniopw
    command: server /data --console-address ":9001"
    ports: ["9000:9000", "9001:9001"]
```

With this, a developer can run `docker-compose up` and have the whole stack running on their machine: - Backend on http://localhost:8000 (with interactive docs at `/docs` if using FastAPI's Swagger UI [16]). - Frontend on http://localhost:3000 (the Next.js app). - MinIO accessible on http://localhost:9001 for a web console (to see stored files, though not necessary). - They can test uploading a contract via the UI and see logs in console for each component.

We ensure the code is hot-reloading in dev (mount volumes or use Next.js live reload).

Deployment (Cloud): For production, we aim to host with minimal cost: - **Frontend:** Deploy the Next.js app to **Vercel** (which offers a generous free tier). Vercel will host the static site and handle getServerSideProps or API routes if we had any (but here our API is separate). We set the env var `NEXT_PUBLIC_API_URL` to the backend's URL. - **Backend & Worker:** We can use **Render.com** or **Railway.app** to host the FastAPI server and the Celery worker. These platforms allow deploying from a Git repo with Docker, and can run background workers. Render, for instance, lets you define a web service and a background worker service on the same repo image. A single small instance (e.g., $7/month) might handle both, but better to separate if possible (the worker might sometimes spike CPU). - Alternatively, deploy on a small **VM (e.g., DigitalOcean droplet or AWS Lightsail)**. A $20/month droplet could run Postgres, backend, worker, and Redis all together for low volume. But for reliability, separating DB is wise. - **Database:** Use a managed Postgres if possible (e.g., Neon.tech has a free Postgres tier, Supabase has free tier, or Railway's Postgres addon). Managed DB prevents data loss and requires less ops. With Terraform, one could set up e.g. an AWS RDS or an Azure Postgres. - **Object Storage:** Could use **Backblaze B2** which is very cheap (and S3-compatible), or a small AWS S3 bucket (cost likely <$0.10/month initially). Or continue using Supabase storage if going that route. - **Secrets:** On Vercel/Render, you can add environment variables via their dashboard (they will be injected into the container). If using Vault in cloud, you would host Vault (maybe on the same VM or a small server) – that might be overkill for MVP. Many opt to store secrets in the cloud platform's config instead. Vault could come into play if the stack grows.

Terraform: We can create Terraform scripts to provision infrastructure. For example: - Terraform AWS provider to create: - An EC2 instance (for app and worker) with Docker installed (using user-data to set it up). - An RDS Postgres instance (small). - An S3 bucket. - IAM roles for the EC2 to access S3 (so we don't even need long-term S3 keys on the box). - Security groups to allow inbound traffic on 80/443 for API if needed (though maybe we put API behind a load balancer or use an ALB). - Alternatively, use Terraform to set up cloud services like Vault, or DNS records for a custom domain.

Because the budget is limited, one might start on free tiers: - Frontend on Vercel (free). - Backend on Railway (free tier might cover small usage). - DB on Supabase (free dev tier up to certain size). - If usage grows, move to paid small plans accordingly. Terraform can be introduced once the pattern is understood, to codify it.

Docker Image Build: We should have Dockerfiles for the backend and frontend. For example, `Dockerfile.backend`:

```
FROM python:3.11-slim
WORKDIR /app
COPY backend/requirements.txt .
RUN pip install -r requirements.txt
COPY backend/. .
ENV PYTHONUNBUFFERED=1
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

and `Dockerfile.frontend`:

```
FROM node:18-alpine AS builder
WORKDIR /app
COPY frontend/package.json frontend/package-lock.json ./
RUN npm ci
```

```
COPY frontend/. .
RUN npm run build

FROM node:18-alpine AS runner
WORKDIR /app
COPY --from=builder /app/.next ./.next
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/package.json ./
ENV PORT=3000
EXPOSE 3000
CMD ["npm", "start"]
```

This builds and then runs the Next.js app in production mode. (Alternatively, since Next pages are mostly static and the dynamic parts call the API, we could even do a static export and serve with a simpler server or on Vercel.)

Monitoring in Production: If using a VM, we'd deploy an agent for monitoring (like a Grafana agent or CloudWatch). If using Render, they have basic metrics and one can forward logs to Papertrail or similar. Given budget, open-source solutions like a combined Prometheus+Grafana on a small instance could be set up (but that might be overkill for start). At minimum, keep error alerting – e.g., integrate Sentry or a simple email on unhandled exceptions, so we get notified of failures.

Domain & SSL: If this is a SaaS, we'll have a domain e.g., `gdprcheck.example.com`. The frontend on Vercel can easily map to a custom domain with SSL. The backend, if on a VM, might need an Nginx reverse proxy with Let's Encrypt certificate. If on Render, it can also attach a custom domain and manage SSL.

Cost Control: The architecture is optimized to stay within budget: - Use free/community editions and small instances. E.g., if we self-host we might run everything on one $40/mo server. If splitting, maybe $10 for DB, $10 for Redis, $20 for app – still ~$40 total. - The major variable cost is LLM API usage. If usage grows, monthly cost could approach the upper range (£400 ~ $500). We can mitigate this by using GPT-3.5 for some checks, or caching results if the same contract is processed multiple times. Or allow an option for users to use their own API key (so the cost is on them). - We avoid expensive dependencies: all components used (FastAPI, Celery, Postgres, Redis, Vault, OpenTelemetry, TipTap, etc.) are open-source or have free tiers. We only pay for cloud resources.

In conclusion, deploying this system involves containerizing the components and using cloud services to host them. Thanks to Next.js and FastAPI, the deployment can be done in a serverless fashion too (e.g., Next.js on Vercel and FastAPI on Azure Container Apps or AWS Fargate). The dev team should be able to spin up the whole stack locally to iterate quickly, then use Terraform or platform configs to mirror that setup in production, ensuring consistency and compliance (e.g., Terraform can enforce security groups, Vault policies, etc., as code). We aim for clarity and reproducibility – any new developer should be able to read the README (or this guide) and get the system running, and understand how data flows securely from upload to report.

Part 2: System Architecture Diagram

Figure: High-level architecture of the Blackletter GDPR Vendor Contract Checker. Data flows from user upload to final report through asynchronous processing pipelines.

1. **User & Frontend:** The user uploads a contract via the Next.js frontend (with TipTap UI for text highlighting). The frontend sends the file and parameters to the backend API over HTTPS.
2. **API Layer (BFF):** The FastAPI (or NestJS) backend receives the file. It stores the file (e.g., in S3 storage) and creates a job record in the PostgreSQL database. It then enqueues an asynchronous processing task to a queue (Celery with Redis/RabbitMQ), and returns a job ID immediately [1] .
3. **Async Queue & Workers:** A Celery worker listening on the queue picks up the job. The processing pipeline begins:
4. **Document Parser:** The worker retrieves the file from storage and extracts text (using PDF/DOCX parsers and OCR if needed). Extracted text is saved to the DB for reference.
5. **Rule Engine:** The text is scanned for GDPR clauses and vague terms using regex rules loaded from JSON/YAML. Findings (missing clauses or flags) are logged to the DB.
6. **Embed & Index:** The text is split into chunks, and each chunk is vectorized via an embedding model. The embeddings are stored in a vector index (pgvector in Postgres, or OpenSearch). This enables semantic lookup of relevant contract snippets.
7. **LLM Judge:** For each compliance question (each rule), relevant text chunks are retrieved (via similarity search) from the vector index [2] . The worker calls the LLM (OpenAI API or other) with the chunk and question, receiving a judgment with explanation. The LLM's analysis is stored, including any citations to the contract text.
8. **Report Builder:** The worker compiles rule results and LLM outputs into a final report (HTML summary and a PDF file). This report is saved (in DB and/or S3). The job status is marked complete in the DB.
9. **Report Delivery:** The frontend polls the API (or gets notified) that processing is complete. The user can then retrieve the results via `GET /review/{job_id}`. The API reads the summary and report path from the DB, and returns the data or a download link for the PDF.
10. **Data Storage & Services:** Throughout the process, data is stored securely:
11. The **PostgreSQL** database holds structured data (job records, rule outcomes, chunk embeddings via pgvector, etc.) with encryption at rest.
12. The **S3 storage** holds the original file and the generated PDF report. All objects are stored in a tenant-isolated path and can be encrypted.
13. **Secrets Vault:** Credentials (DB connection strings, API keys) are fetched from Vault or secure env vars by the API and workers [6] . This ensures no hard-coded secrets in code or images.
14. **Observability:** Both the API and workers are instrumented with OpenTelemetry (dotted lines to Monitoring). Traces and logs are sent to a monitoring system (like Jaeger or Grafana) for debugging and auditing. No sensitive contract data is included in logs, only high-level events.
15. **Security & Privacy:** The entire flow implements privacy-by-design. Only small chunks of text are sent to the LLM (never the whole document), and personal data can be redacted before LLM processing. All data is processed within isolated tenant context – an authenticated user can only access their own jobs (enforced in the API and DB queries). The system supports delete-on-request and auto-deletion of data after a retention period (e.g., 30 days) [6] , ensuring no lingering personal data.

Overall, this architecture leverages asynchronous, event-driven processing (via the task queue) and a combination of rule-based and AI-based components to deliver a comprehensive GDPR compliance report. Components communicate through well-defined interfaces (HTTP API, message queue, database queries), making the system modular and scalable while keeping costs and resource usage in check for an MVP deployment.

---

[1] [12] [13] [14] [15] I Built an Open-Source RAG API for Docs, GitHub Issues and READMEs : r/LangChain
https://www.reddit.com/r/LangChain/comments/1i89s54/i_built_an_opensource_rag_api_for_docs_github/

[2] [5] [8] PostgreSQL as a Vector Database: A Pgvector Tutorial | TigerData
https://www.tigerdata.com/blog/postgresql-as-a-vector-database-using-pgvector

[3] [9] [10] Anthropic-Style Citations with Any LLM | by Michael Ryaboy | Data Science Collective | Medium
https://medium.com/data-science-collective/anthropic-style-citations-with-any-llm-2c061671ddd5

[4] [6] [11] [16] Blackletter Systems – Full Build Plan (unified).pdf
file://file-EUpJySpexvNtX91mRRcZhb

[7] What needs to be included in the contract? | ICO
https://ico.org.uk/for-organisations/uk-gdpr-guidance-and-resources/accountability-and-governance/contracts-and-liabilities-between-controllers-and-processors-multi/what-needs-to-be-included-in-the-contract/