ChatGPT

LLM Judge and Report Builder Modules for GDPR Contract Review

Introduction

Blackletter Systems' contract review tool leverages AI to evaluate legal documents for GDPR compliance. We design two core components: an **LLM Judge** module that uses large language models to analyze contract clauses against GDPR rules, and a **Report Builder** module that compiles the findings into a professional, branded report. Each module is implemented as a standalone, reusable Python package with clear interfaces, thorough typing, and documentation, ready to integrate into a FastAPI + Celery pipeline. The LLM Judge acts as an intelligent *scorer* of compliance (producing structured evaluations), while the Report Builder is a *renderer* that transforms those evaluations into human-friendly PDF/HTML reports. This separation of concerns ensures maintainability and clarity in the system design.

Below, we detail the design and implementation of each module, including their responsibilities, internal architecture, and how they meet the requirements (like deterministic LLM outputs, guardrails against hallucinations, batch processing, and report formatting). We also include Python code snippets (with docstrings and type hints) to illustrate how these modules can be implemented.

LLM Judge Module

Purpose: The LLM Judge evaluates extracted contract clause findings against GDPR compliance rules using large language models (LLMs). It takes in clause data plus a specific rule, and produces a structured JSON output assessing compliance (verdict, risk level, rationale, etc.). This module automates what a human legal analyst might do when reviewing a contract clause for GDPR adherence, by following a pre-defined rubric for each type of compliance issue.

Input and Output Schema

The LLM Judge expects input in a JSON schema that provides all necessary context for evaluation. The input includes: - `rule` – an identifier or description of the GDPR compliance rule in question. For example, this could be a reference like `"GDPR Article 5 - Data Minimization"` or a specific check like `"Missing Data Processing Agreement clause"`.
- `snippet` – the actual text of the contract clause or excerpt that was identified as relevant to the rule. This is the piece of contract language the LLM will analyze. - `context` – optional additional context, such as surrounding contract text or a summary of the contract, which might help the LLM understand the broader setting. This could also include the full text of the contract or relevant sections for more context. - `citations` – any references or source citations associated with the rule or snippet. For instance, this might include the source of the rule (like the GDPR text) or location info for the snippet (e.g., "Clause 12.3 of Data Processing Agreement"). These help the LLM ground its analysis and can be included in the prompt for completeness.

The output from the LLM Judge is a JSON object with the following fields: - `verdict` – a concise classification of compliance for that rule, e.g. `"Compliant"`, `"Non-compliant"`, `"Partially compliant"`, or `"Not found"`. This is the overall judgment. - `risk_level` – a rating of the severity

or risk associated with this finding. For example, it might be `"Low"`, `"Medium"`, `"High"` corresponding to the level of GDPR compliance risk. (This can also be encoded as colors or numbers internally, but ultimately will feed into a traffic-light style indicator in the report.) - `rationale` – a textual explanation justifying the verdict. This should reference the clause content and the rule requirements, explaining why the clause is or isn't compliant. The rationale is ideally backed by evidence (quotes from the snippet or GDPR text) to avoid unsupported statements. It's crucial that this rationale **only uses information from the provided snippet/context** and does not introduce new "hallucinated" facts [1] [2]. - `improvement_suggestions` – a list of suggestions for how to improve the clause if it's non-compliant or partially compliant. Each suggestion might be a short actionable tip. For example, *"Include an explicit data retention period to comply with Article 5(1)(e)"*. These suggestions may come from an internal playbook (for known issues) or be generated by the LLM based on best practices. (Note: The module will **not** directly rewrite contract text unless explicitly configured to do "redlining" – see guardrails below.) - `direct_quotes` – a list of direct quotes extracted from the snippet or official GDPR guidance that support the rationale. For instance, if the rule is about breach notification timeframe and the clause is missing it, a direct quote might be from the GDPR text: *"'the controller shall notify the personal data breach to the supervisory authority without undue delay and, where feasible, not later than 72 hours' (GDPR Art.33)"*. Including these direct quotes ensures traceability and shows the specific text that influenced the verdict.

This structured output makes it easy for downstream components (like the report builder) to consume the results. Below is an example Python data class defining the input and output schema for clarity:

```
from dataclasses import dataclass
from typing import List, Optional

@dataclass
class ClauseFinding:
    rule: str
    snippet: str
    context: Optional[str] = None
    citations: Optional[List[str]] = None

@dataclass
class ClauseJudgment:
    verdict: str
    risk_level: str
    rationale: str
    improvement_suggestions: List[str]
    direct_quotes: List[str]
    # We can also include metadata like model info or prompt ID here if
needed.
```

Prompt Templates per Issue Type

Not all compliance issues are evaluated the same way. The **LLM Judge uses templated prompts tailored to the type of issue/rule** being checked. For example: - If the issue type is **"vague term"** (the rule might be "Avoid ambiguous terms in data handling clauses"), the prompt should instruct the LLM to check if the `snippet` contains unclear or undefined terms and explain why they are problematic. - If the issue type is **"missing clause"** (e.g., "GDPR Article 28 requires a Data Processing Agreement

clause"), the prompt should guide the LLM to determine if the necessary clause is present in the snippet or not, and what the impact is if it's missing.

We maintain a library of prompt templates indexed by issue type or rule. These templates are essentially fill-in-the-blank texts that incorporate the input JSON fields. For example, a prompt template for a missing clause might look like:

```
"You are a legal AI reviewing a contract for GDPR compliance. The specific
rule is: {rule}.

Here is an excerpt from the contract:
\"\"\"
{snippet}
\"\"\"

Context: {context}

Determine if this contract excerpt satisfies the requirements of the rule.
- If yes, output that it's compliant and why.
- If not, output that it's not compliant and explain what is missing.
Provide your answer in JSON with the following keys: verdict, risk_level,
rationale, improvement_suggestions, direct_quotes."
```

Each template ends by explicitly asking the model to output JSON conforming to our schema. By doing this, we **constrain the LLM to produce a structured output**. We ensure each field is addressed. The templates may vary: for a "vague term" check, the prompt might say, *"Identify any vague or undefined terms in the clause and explain why they create compliance uncertainty"*, guiding the model to focus on that aspect.

These templates are versioned and stored (perhaps in a database or YAML files) and can be selected dynamically based on the input. Each template has an identifier (`prompt_template_id`) which we will include in the trace metadata for debugging and observability.

Deterministic LLM Decoding Settings

When the LLM Judge calls the LLM (such as GPT-4), it uses **deterministic decoding settings** to ensure consistent outputs for the same input. Specifically, we set a low temperature and top-p: - `temperature = 0` (or very close to 0) to minimize randomness in the model's output. - `top_p = 0.1` to further constrain the token sampling to the top probability mass [3].

Using temperature 0 and a small top-p effectively makes the generation process deterministic, meaning the model will almost always pick the highest-probability answer for a given prompt [3]. This is important for a compliance tool – we want the judgments to be repeatable and not vary from run to run. (Note: In practice GPT-4 with temp=0 is *nearly* deterministic, though there can be minor non-determinism in rare cases due to tokenization; but this configuration gets us as close as possible [4].)

Additionally, we may use the model's **"function calling" or JSON mode** if available, or post-process the raw output to ensure it's valid JSON. The system could, for example, verify that the output can be parsed as JSON. If not, it might retry or apply a fix (like wrapping the output or adding missing braces).

Ensuring a valid JSON output 100% of the time is crucial for downstream automation. In summary, by carefully designing prompts and using strict decoding settings, we aim for consistent, parseable, and correct JSON outputs from the LLM.

Primary Model and Fallbacks (Adapter Pattern)

The primary LLM used is **OpenAI's GPT-4**, given its strong performance with complex instructions. However, we design the module to be model-agnostic via an **adapter pattern** so that other models like **Anthropic Claude** or **Google Gemini** can be used as fallbacks or in different deployments.

In practice, this means we have an abstraction like a `LLMClient` or `ModelProvider` class. This class can be configured with a provider type (`"openai"`, `"anthropic"`, `"google"`, etc.) and appropriate credentials (API keys) and settings. The interface of the LLMClient might have a method `generate(prompt) -> str` that hides the details of API calls. For example:

```
class LLMClient:
    def __init__(self, provider: str, model_name: str, api_key: str, **opts):
        # Initialize with provider (openai, anthropic, etc.)
        # and model name (e.g., "gpt-4", "claude-v2")
        self.provider = provider
        self.model_name = model_name
        self.api_key = api_key
        self.opts = opts  # additional options like temperature, etc.
        # ... setup client as needed ...

    def generate(self, prompt: str) -> str:
        """Generate completion using the specified LLM provider."""
        if self.provider == "openai":
            # Use OpenAI's API (pseudo-code)
            import openai
            response = openai.ChatCompletion.create(
                model=self.model_name,
                messages=[{"role": "user", "content": prompt}],
                temperature=self.opts.get("temperature", 0),
                top_p=self.opts.get("top_p", 1)
            )
            return response.choices[0].message.content
        elif self.provider == "anthropic":
            # Use Anthropic's API (pseudo-code)
            import anthropic
            client = anthropic.Client(api_key=self.api_key)
            response = client.completion(
                prompt=prompt,
                model=self.model_name,
                max_tokens=1000,
                temperature=self.opts.get("temperature", 0)
            )
            return response.get("completion")
        # ... other providers like Google Gemini could be added similarly ...
```

```
        else:
            raise ValueError(f"Unsupported LLM provider: {self.provider}")
```

By using this adapter approach, switching from GPT-4 to Claude or another model is straightforward – the rest of the Judge module's logic (prompt assembly, output parsing) remains the same. This is valuable for fallbacks: if a call to GPT-4 fails or times out, the system can automatically retry with Claude, for instance. Or if running on-premises, a firm might prefer a self-hosted LLM. Our design cleanly supports these scenarios [5] (e.g., a PyPI library for model adapters notes support for OpenAI, Google Gemini, and Anthropic Claude all under a unified interface [5]).

The default configuration will use OpenAI's GPT-4 model, with, say, Anthropic Claude 2 as a backup. The selection could be dynamic: e.g., if an issue type is better handled by a particular model (perhaps some models fine-tuned on legal text), the adapter can route accordingly. All such decisions are abstracted behind the LLMClient.

Guardrails for Accuracy and Safety

Because we are using an LLM in a legal compliance context, we must implement **guardrails** to ensure the output is reliable, secure, and within scope. Key guardrails include: - **Prompt Sanitization:** Any user-supplied text (like the contract snippet or context) is sanitized before being inserted into the prompt. This includes removing or escaping problematic characters or instructions that could lead to prompt injection attacks [6] . For instance, if the contract text contained something like "`<|im_start|>system`", we would neutralize it to prevent it from manipulating the LLM's role structure. We also ensure that no personally identifiable information (PII) or sensitive data leaks via the prompt (unless necessary for analysis, and even then it might be masked or generalized). The AWS Prescriptive guidance on LLM safety explicitly suggests input validation and prompt sanitization as safeguards [6] . Our module will provide utility functions to clean inputs (removing HTML/SQL or special tokens, etc.) before prompt assembly. - **No Hallucinations – Cite Only Provided Text:** The Judge's responses must be grounded in the snippet and known GDPR rules, not the model's general knowledge. We instruct the model (in the prompt) to **only rely on the given text and rule**. For example, the prompt might say: *"Only use the contract excerpt and the GDPR rule provided. Do not introduce facts that are not present in the provided text. If the excerpt is insufficient, state that explicitly."* We further enforce this by requiring **direct quotes** in the output rationale. The model knows it should pull exact phrases from the snippet or the rule text to support its conclusions, which naturally curbs hallucinations [1] . In essence, the model is acting in a *closed-book* setting with only the supplied references. This approach aligns with retrieval-augmented generation practices, where the LLM's credibility comes from citing source text [1] . Any content not traceable to the snippet or official rule is not to be included. - **No Unrequested Clause Rewriting:** The Judge's role is to evaluate and suggest improvements, **not to rewrite contract language unless explicitly asked**. Rewriting clauses (like giving a fixed version of the contract text) could raise liability issues or deviate from the lawyer's drafting style. Therefore, by default, our prompts and instructions tell the LLM *not* to produce full rewrites. Instead, it should say e.g., "Recommend adding a clause about X" rather than writing the clause itself. Only if a specific rule or configuration indicates that "redlining is enabled" (meaning we want the AI to draft a revised clause) would we allow or prompt the model to output a rewritten version. This guardrail prevents the model from going beyond its evaluative mandate. - **Size Limits and Chaining:** If the context or snippet is very large (contracts can be dozens of pages), the module might chunk the text or summarize it before sending to the LLM, to avoid context window limits. Alternatively, the extraction of snippet is assumed to give manageable chunks. In cases where multiple clauses relate to one rule, we could either judge them separately or combine them carefully. The module includes logic to handle or warn about overly long inputs. - **Monitoring and Fallbacks:** The system monitors for any sign the model output is not

following format (e.g., not valid JSON or missing fields). If such a case occurs, the Judge can retry with a more constrained prompt or use a simpler model. Also, all outputs can be subject to a post-hoc validation (for example, ensuring that if verdict is "Compliant" then risk_level is "Low" or similar consistency checks). - **Rate limiting and API errors:** The Judge module, being part of a pipeline, is designed to handle exceptions gracefully. If the LLM API call fails due to rate limits or network issues, the module will implement a retry with backoff. Integration with Celery (or NATS) means such errors could also cause the task to be retried. This ensures robustness in production.

Overall, these guardrails ensure the **LLM Judge is reliable and secure** – it should behave like a focused legal assistant that only evaluates based on given evidence and never goes off-script.

Traceability and Metadata

For each judgment the LLM makes, the module emits **structured trace metadata**. This metadata isn't necessarily part of the output shown to end-users, but it is logged/stored for debugging, audit, or analysis. The trace data can include: - The model used (e.g., `"gpt-4"` or `"claude-2"`). - The provider (e.g., OpenAI, Anthropic). - The `prompt_template_id` that was applied (so we know which prompt phrasing was used). - Timestamps (start and end time of the API call, and maybe a unique request ID). - Token usage or costs if available (for monitoring usage of the AI API). - Possibly a summary of input (like a hash of the snippet or first 100 chars) to quickly identify which clause was being judged.

This trace metadata is invaluable when reviewing how a particular verdict was reached or when troubleshooting the LLM's performance on certain rules. There are emerging tools and standards for LLM observability that could be integrated; for instance, **Langfuse or OpenTelemetry tracing for LLMs** could record these interactions [7]. In our implementation, after getting the result from the LLM, we might attach a `trace` object alongside the judgment result. For example:

```
judgment = {
    "verdict": verdict,
    "risk_level": risk_level,
    "rationale": rationale,
    "improvement_suggestions": suggestions,
    "direct_quotes": quotes
}
trace = {
    "model": self.model_name,
    "provider": self.provider,
    "prompt_template_id": template_id,
    "timestamp": datetime.utcnow().isoformat()
}
return judgment, trace
```

The trace can be logged to a database or file. This way, if a lawyer ever questions *"How did the AI come up with this?"*, we can refer back to the exact prompt and model output, ensuring transparency.

Batch Processing and Queue Integration

The LLM Judge is built to handle **batch processing**, which is crucial when reviewing an entire contract (dozens of clauses) asynchronously. The module can be invoked on one clause at a time, but we also

support batch mode where a list of clause findings is evaluated in one go. For example, a function `judge_clauses(list_of_findings)` will iterate or map over them, possibly in parallel.

In a production scenario, this ties into Celery tasks or NATS message queues. The system might extract all relevant snippets from a contract, enqueue each as a task for the Judge to evaluate, and then collect the results. Celery is a natural fit here – it is a distributed task queue that can handle a large number of background jobs and retries [8]. Each clause evaluation can be a Celery task (with the JSON input as payload), and Celery workers (running our LLM Judge code) will pick tasks from the queue and execute them concurrently [8]. We ensure the LLM calls themselves are rate-limited (to avoid hitting API limits) either by Celery concurrency settings or internal throttling.

NATS can also be used (either as Celery's broker or as an alternative work queue). NATS JetStream or a simple queue subscription could distribute clause evaluation tasks among workers. The LLM Judge module is indifferent to the queue technology – it just exposes a function that processes one input, so as long as the worker calls that function with the task data, it works. Our design documentation will include examples of using the module in a Celery task, e.g.:

```
# Pseudo-code for a Celery task using the LLM Judge
from celery import Celery
from mypackage.llm_judge import LLMJudge, ClauseFinding

app = Celery('gdpr_judge')
judge = LLMJudge(model_provider="openai", model_name="gpt-4")  # configured
globally

@app.task(name="judge_clause")
def judge_clause_task(finding: dict):
    # Convert dict to ClauseFinding dataclass
    cf = ClauseFinding(**finding)
    result, trace = judge.evaluate_clause(cf)
    # Here we might save result to DB or return it
    return {"result": result, "trace": trace}
```

Such tasks can be queued in bulk, and Celery's result backend can collect them. We also implement **batch mode** where a list of findings can be passed to one method for sequential processing – useful for synchronous use or testing.

Code Implementation of LLM Judge

Finally, to illustrate the module's implementation, below is a simplified version of what the **LLM Judge** class could look like in code. This snippet brings together many of the points discussed: it uses an LLM client, applies a template, sanitizes inputs, and parses the LLM output into our structured format. In practice, this would be part of a Python package (e.g., `blackletter_llm_judge`) with full docstrings and type annotations:

```
from typing import List, Dict, Any
import json
import datetime
```

```
class LLMJudge:
    """
    LLMJudge evaluates contract clause findings against compliance rules
using an LLM.
    It produces a structured verdict with risk level, rationale, suggestions,
and quotes.
    """
    def __init__(self, model_provider: str = "openai", model_name: str =
"gpt-4", api_key: str = "",
                 prompt_templates: Dict[str, str] = None):
        """
        Initialize the LLMJudge with a model provider (e.g. 'openai',
'anthropic'),
        model name, API credentials, and optional custom prompt templates.
        """
        self.model_provider = model_provider
        self.model_name = model_name
        self.api_key = api_key
        self.prompt_templates = prompt_templates or {}
        # Initialize the LLM client/adapter
        self.client = LLMClient(provider=model_provider,
model_name=model_name, api_key=api_key,
                                temperature=0, top_p=0.1)

    def sanitize_text(self, text: str) -> str:
        """
        Basic prompt sanitization to neutralize problematic tokens or
content.
        This might remove harmful instructions or ensure no prompt injection.
        """
        if text is None:
            return ""
        # Example sanitization: strip control characters, special sequences
        forbidden_sequences = ["<|im_start|>", "<|im_end|>"]
        clean = text
        for seq in forbidden_sequences:
            clean = clean.replace(seq, "")
        return clean

    def evaluate_clause(self, finding: ClauseFinding) -> Dict[str, Any]:
        """
        Evaluate a single clause finding against its rule using the LLM.
        Returns a dict with verdict, risk_level, rationale,
improvement_suggestions, direct_quotes.
        """
        # Sanitize inputs
        rule = self.sanitize_text(finding.rule)
        snippet = self.sanitize_text(finding.snippet)
        context = self.sanitize_text(finding.context or "")
        # Choose appropriate prompt template
        issue_type = self._determine_issue_type(finding.rule)
```

```
        template = self.prompt_templates.get(issue_type) or
self._default_template()
        # Fill in the template
        prompt = template.format(rule=rule, snippet=snippet, context=context)
        # Call the LLM (deterministic decoding is set in the LLMClient
initialization)
        raw_output = self.client.generate(prompt)
        # Parse the LLM output. Assuming it's JSON, but if it's a JSON
string, parse it.
        try:
            output = json.loads(raw_output)
        except json.JSONDecodeError:
            # If the model didn't return clean JSON (it should), attempt to
extract JSON
            # For simplicity, here we do a naive fix or re-prompt the model
to correct format
            output_str = raw_output.strip().strip("```")  # remove markdown
if any
            left_brace = output_str.find('{')
            right_brace = output_str.rfind('}')
            if left_brace != -1 and right_brace != -1:
                json_str = output_str[left_brace:right_brace+1]
                output = json.loads(json_str)
            else:
                raise  # can't fix easily
        # Attach trace metadata
        output["trace"] = {
            "model": self.model_name,
            "provider": self.model_provider,
            "prompt_template_id": issue_type,
            "timestamp": datetime.datetime.utcnow().isoformat()
        }
        return output

    def _determine_issue_type(self, rule: str) -> str:
        """
        Determine issue type (e.g., 'vague_term', 'missing_clause') from the
rule description.
        This can be a simple mapping or regex check on the rule text.
        """
        text = rule.lower()
        if "missing" in text or "not present" in text:
            return "missing_clause"
        if "vague" in text or "ambiguous" in text:
            return "vague_term"
        # ... more rules ...
        return "general_compliance"

    def _default_template(self) -> str:
        """Fallback prompt template if a specific one is not found."""
        return (
```

```
            "You are a legal AI checking GDPR compliance.\n"
            "Rule: {rule}\nContract Excerpt:\n\"\"\"\n{snippet}
\n\"\"\"\n{context}\n"
            "Evaluate the excerpt against the rule. "
            "Provide a JSON with keys verdict, risk_level, rationale,
improvement_suggestions, direct_quotes."
        )
```

In this code, we see: - `LLMJudge.__init__`: sets up the LLM client with deterministic parameters. Also, it accepts a `prompt_templates` dictionary, which can be loaded from an external file or defined in code. - `sanitize_text`: a simple method to strip out or replace any known dangerous tokens or patterns. (In a real system, this would be more comprehensive, possibly using regex to remove things like `\">` or other prompt injection markers [6].) - `evaluate_clause`: the core method that fills in the prompt and invokes the LLM. It also handles JSON parsing carefully. It uses `_determine_issue_type` to pick a prompt template. The `issue_type` doubles as a `prompt_template_id` here, which we include in the trace. - **JSON parsing**: We try to load the JSON. In case the model returns JSON wrapped in markdown or with extra text, we attempt to strip extraneous parts. (We expect with well-crafted prompts and GPT-4's format capabilities, this will be rare.) - We attach a `"trace"` field into the output with metadata. (Alternatively, we might return it separately; but including in output can be convenient for testing. In production, we might log it instead.) - **Issue type detection** is rudimentary here (based on keywords in rule text). In practice, the rule input might include an explicit type or code, so we'd use that. Or the prompt template selection logic can be more direct (e.g., a mapping from a rule ID to template).

This class would be part of a package and include comprehensive docstrings explaining each method's purpose, parameters, and return types (as shown). The functions are all type-annotated for clarity. The design is ready for production integration: the `LLMJudge` could be initialized at application startup (e.g., as a FastAPI dependency or a Celery worker global) and used to process tasks as needed.

Report Builder Module

Purpose: The Report Builder takes the structured outputs from the LLM Judge (the issues and their evaluations) and **constructs a polished report** in HTML/PDF format. This report is meant for end-users (lawyers, compliance officers, or clients) to review the GDPR compliance findings. It includes summaries, detailed findings per rule, suggestions for remediation, and references. Essentially, it translates the raw JSON results into a human-readable document, with branding and helpful organization.

Input Data and Workflow

The input to the Report Builder is a **list of issue objects**, which contain the fields produced by the Judge (verdict, risk, rationale, etc.) along with any additional metadata needed for reporting. In the broader system, this likely corresponds to the "frontend schema" for issues. Each issue object might include: - A reference to the **GDPR Article or Rule** (e.g., Article 5(1)(b) – Purpose Limitation). - The **clause text snippet** itself (or an identifier for it) – possibly included if we want to show the actual contract text in the report for context. - The structured evaluation fields (verdict, risk_level, rationale, improvement_suggestions, direct_quotes) as discussed. - Possibly a classification of issue type or severity that can be used for grouping or sorting.

The Report Builder will typically be invoked after all clause evaluations are done. The high-level workflow is: 1. The issues (judgments) are gathered, likely from a database or in-memory after the

Judge module finishes. 2. The Report Builder organizes these issues by categories (for example, grouping by GDPR Article number or by risk level). 3. It then populates an **HTML template** or programmatically creates HTML sections corresponding to various parts of the report. 4. If a **JSON output** is needed (for an API or for the front-end), it may also produce a JSON representation of the full report (for instance, a JSON that has sections and issues, which the front-end could render). But the primary goal here is to produce a **formatted PDF** for delivery. 5. The HTML is converted to PDF using a library or tool like **WeasyPrint or wkhtmltopdf** [9]. 6. The PDF (and/or HTML) is then saved to an object store or storage service and possibly a reference (URL or path) is attached to the job record for later retrieval/download.

Report Sections and Organization

The output report is divided into clear sections to make it digestible. We plan the following sections:

- **Cover Page / Title:** If branding is needed, a cover page with the firm's logo, the title of the report (e.g., "GDPR Contract Compliance Report"), date, and the client/contract name could be included. (Branding is discussed later.)

- **Executive Summary:** A high-level overview of the findings. This includes key **KPI stats** such as:

- Total number of issues found.
- Count of issues by severity (e.g., how many high-risk, medium-risk, low-risk findings).
- Possibly a percentage of compliance (for example, if out of 10 checked rules, 7 were compliant and 3 had issues, we might say "70% of checked items are compliant").
- A short textual summary highlighting major concerns (e.g., "The contract lacks a proper data breach notification clause and does not specify data retention limits, which are high-risk issues.").

- A **traffic-light chart or RAG status graphic** indicating overall risk. For instance, we could present a simple colored indicator: if there are high-risk items, the overall might be Red; if only low-risk issues, maybe Green. Often reports use Red/Amber/Green to signal risk levels [10]. We might include a small table or list: ▉ High: 2 issues, ▒ Medium: 3 issues, ▧ Low: 5 issues. This quick visual cue helps readers grasp the situation at a glance [10].

- **Rule-by-Rule Findings (Grouped by GDPR Article):** This section forms the bulk of the report. We list each GDPR requirement that was evaluated, grouped by Article or section of the GDPR:

- For example, under **"Article 5 – Principles relating to processing of personal data"**, we might have subitems for each relevant principle like (a) lawful, fair, transparent, (b) purpose limitation, (c) data minimization, etc. For each, we state whether the contract meets the requirement.
- Under **"Article 28 – Processor Obligations"**, we might list items like whether a Data Processing Agreement exists, whether sub-processor approval is mentioned, etc.
- Each item will show the **Rule/Requirement name**, the **Verdict** (compliant or not), and the **Rationale**. We will often include the **contract snippet** (or a portion of it) in a blockquote or indented area, so the reader sees the exact text that was evaluated. Then, following that, the rationale explains the finding, and if not compliant, what is missing or why it's inadequate (drawn from the LLM's rationale, possibly edited or formatted for readability).

- **Direct Quotes/Citations:** If the LLM provided direct quotes (like excerpts from GDPR text or the contract), these can be formatted as footnotes or parenthetical citations. For instance, if the rationale says "This fails to specify a retention period as required by GDPR", we might footnote:

GDPR Art.5(1)(e) and in a footnote section or tooltip show the actual text from GDPR that was quoted. Providing these citations builds trust, as readers can verify the source of the rule [1].

- **Remediation Suggestions:** After listing all findings, we include a section focused on **Recommendations**. This takes the improvement suggestions from each issue and groups them into a coherent action plan. We might structure it as:

- High-risk issues first: For each, a suggestion on how to remediate. E.g., "Add a clause requiring breach notification to the controller within 72 hours of discovery of a breach (to satisfy GDPR Art. 33)."
- Medium-risk next, etc. Or group by article again.
- If we have an internal **YAML playbook** of common remediation steps for known issues, the Report Builder can cross-reference it. For example, if the LLM flagged a missing Article 30 record-keeping clause, and our playbook has a canned recommendation for that, we might use the playbook text instead of or in addition to the LLM's suggestion. The module could be configured to prefer playbook suggestions where available (for consistency), falling back to LLM suggestions otherwise.

- This section essentially answers: *"What should we do to fix or improve this contract?"* It should be actionable. We might format it as a list of bullet points or a table of Issue -> Recommendation.

- **Optional Redline Drafting:** If the system is configured with a "redlining" feature (where the AI can propose contract language changes), the report can include an appendix with **Drafted Clauses or Redlined Edits**. For each major issue, we could have the AI actually suggest a rewritten clause or an insertion. This would only appear if the user enabled it (since not all law firms will be comfortable with AI-generated contract text without review). If included, we label it clearly e.g., "Draft Clause (for review):" followed by a markdown or diff of the suggested text. It's likely we'd present it as a suggestion, not as final wording. This can save lawyers time by giving them a starting point for fixes.

- **Appendices (Optional):** If needed, an appendix could list the full text of relevant GDPR articles (since the report references them) for convenience. Or a glossary of terms. Also, a section on **Methodology** could briefly describe that "This report was generated by Blackletter Systems' AI based on an automated analysis of the contract. All conclusions should be reviewed by a qualified attorney before action." – basically a disclaimer and explanation to manage expectations.

The report structure should be customizable to some extent (via templates or settings) to meet different law firm preferences. But the above outlines a default comprehensive structure.

HTML Generation and PDF Conversion

We generate an **HTML representation** of the report first, because HTML/CSS is a convenient way to format text, apply styles (branding), and ensure the output looks good (either in a browser or converted to PDF). Using HTML allows leveraging existing web design tools for layout and style (for example, we can use CSS for color-coding risk levels, or include the firm's brand styles).

We have two main options to convert HTML to PDF in Python: - **WeasyPrint** – a Python library that can take HTML/CSS and render it to PDF entirely in Python. WeasyPrint is known for being easy to use: "It takes HTML and CSS, and converts it to a usable and potentially beautiful PDF document." [9]. WeasyPrint handles most CSS and even has support for printing features (like page breaks, footers,

etc.). - **wkhtmltopdf (with pdfkit)** – a system tool that uses the WebKit rendering engine to convert HTML to PDF. In Python, `pdfkit` is often used as a wrapper around wkhtmltopdf. This requires that wkhtmltopdf is installed on the server. It's also a reliable method and can handle complex HTML/JS (though JS isn't usually needed for static reports).

For a pure Python approach, we lean towards WeasyPrint. The Report Builder can use WeasyPrint's API to generate PDF bytes from the HTML string. For example:

```
from weasyprint import HTML, CSS

html_content = build_html_report(issues, branding_info)  # a function to
construct the HTML string
# Assuming we have some CSS for styling, either embedded or separate:
css = CSS(string="""
    .high-risk { color: red; font-weight: bold; }
    .medium-risk { color: orange; font-weight: bold; }
    .low-risk { color: green; font-weight: bold; }
    /* ... other styles ... */
""")
pdf_bytes = HTML(string=html_content).write_pdf(stylesheets=[css])
# Now pdf_bytes contains the binary PDF data.
```

In this snippet, `build_html_report` is a function (which we will outline shortly) that returns the full HTML as a string. WeasyPrint's `HTML.write_pdf()` converts it to PDF bytes that we can then write to a file or send over a network. We could also use the WeasyPrint command-line, but using it as a library is straightforward for integration [11] [12].

If we choose wkhtmltopdf, the code might look like:

```
import pdfkit
pdfkit.from_string(html_content, "output.pdf")
```

This will call out to wkhtmltopdf. We need to ensure wkhtmltopdf is installed in the environment. Both methods are viable; we can even support both via configuration (some deployments might prefer not to install wkhtmltopdf, others might already use it).

One advantage of generating an HTML first is we can easily offer an **HTML preview** of the report in a web dashboard before the PDF is downloaded. The PDF ensures a fixed-format for emailing or archiving.

Additionally, **Markdown to HTML conversion** can be supported if needed. For example, if some parts of the content (like suggestions or rationale) are in Markdown format (perhaps authored in a playbook as Markdown), we can use a library like Python-Markdown to convert them to HTML [13]. Python-Markdown is a popular library that "allows users to convert Markdown files into HTML" [13]. We could integrate it so that any field that contains markdown (like a suggestion with bullet points) is rendered properly in the HTML. This gives flexibility in authoring templates or playbook content in a more readable format.

Branding and Customization

Law firms using this tool will likely want the reports to look like their own. Thus, the Report Builder supports branding: - **Logo and Firm Name:** The module can accept a path or URL to the firm's logo image. The HTML template can then place this in the header or cover page. For example, an `<img src="logo.png" alt="Firm Logo" style="height:50px">` could appear on the first page or in a header on each page. The firm's name and maybe tagline can be inserted as well. These can be provided via a config or parameters when calling the report builder. - **Color Scheme:** The CSS can be adjusted to match the firm's branding (e.g., using the firm's official colors for headings or accents). If not specified, a default style with neutral professional look will be used. - **Footer:** Often reports have a footer with page numbers and maybe a confidentiality notice or copyright. WeasyPrint allows setting up page footers via CSS `@page` rules. For instance, a footer might say "Confidential - for internal use only - generated on 2025-08-24". The builder can automatically insert the generation date and job ID in the footer. - **Cover Page Text:** Possibly customizable with project name or client name. E.g., "GDPR Review for Contract: [Contract Name]". We can allow placeholders in our template that the calling code fills in (like contract title, client, date, etc.). - **Jurisdiction and Playbook Config:** The module supports **jurisdiction-aware filtering**, meaning if a law firm only cares about UK-specific GDPR provisions or has some custom rules, the input issues can be filtered or labeled by jurisdiction and the report can present only relevant ones. Also, a configuration might specify which rules to include or exclude in the report (some clients might not need certain sections). - **Language**: While this is targeted at UK firms (thus presumably English reports), the structure allows for localization if needed in the future (by translating section titles, etc., if integrating in EU languages).

We design the Report Builder so that these customizations are done via either template files or parameters, rather than hard-coding values. For example, we might use Jinja2 templating for the HTML, where a template file contains placeholders for logo URL, firm name, etc., and our code renders it with the given context data.

Saving and Exporting the Report

Once the PDF is generated, the module should save it and make it accessible: - We may simply write the PDF to a file path on the server (e.g., in a designated directory or a temporary file). - More robustly, we can upload it to an **object store** (such as AWS S3 or Azure Blob Storage) and then store the returned file URL in our system. That way, the front-end or the user can download it from a cloud location. This is especially useful if the API server is stateless or if we want to offload file serving. For instance, using boto3 to put the file in S3 and get a presigned URL. - The module likely receives a job or report identifier so it knows where to attach the result. For example, we might pass a `job_id` into the report builder, and it will name the PDF as `{job_id}_report.pdf` and update a database record for that job with the file location. - After storing, it could also return the path/URL for immediate use.

All these steps (build HTML, convert to PDF, save file, update record) can be encapsulated in a single high-level function, e.g., `generate_report(issues, config) -> str` (where return is the file path or URL). This function can then be called by a Celery task or a FastAPI endpoint when the analysis is complete.

Example Implementation of Report Builder

Below is a pseudocode example of how the **ReportBuilder** class might be implemented. It outlines how we assemble the HTML and handle PDF conversion and storage:

```
from typing import List, Dict
from weasyprint import HTML, CSS
import markdown
import datetime
import uuid
import os

class ReportBuilder:
    """
    ReportBuilder converts a list of issue objects into a branded GDPR
compliance report (HTML and PDF).
    """
    def __init__(self, template_path: str = None, css_path: str = None,
branding: Dict[str, str] = None):
        """
        Initialize the ReportBuilder with optional HTML template and CSS for
styling.
        branding: dict can include 'firm_name', 'logo_path', 'footer_text',
etc.
        """
        self.template_path = template_path  # Path to an HTML template file
(Jinja2 or simple placeholders)
        self.css_path = css_path  # Path to a CSS file for styling
        self.branding = branding or {}

    def build_html(self, issues: List[Dict]) -> str:
        """
        Construct the HTML report from the issues. If a template is provided,
use it; otherwise build programmatically.
        """
        # If using a template engine like Jinja2:
        if self.template_path:
            from jinja2 import Template
            template_text = open(self.template_path).read()
            template = Template(template_text)
            html = template.render(issues=issues, branding=self.branding,
generated_date=datetime.date.today())
            return html

        # Build HTML manually (simplified)
        firm_name = self.branding.get("firm_name", "Law Firm")
        title = f"{firm_name} – GDPR Contract Review Report"
        html_parts = [f"<html><head><title>{title}</title>"]
        # Basic styling (if no css_path provided)
        html_parts.append("<style>" + self._default_css() + "</style>" if not
self.css_path else
                          f"<link rel='stylesheet' href='{self.css_path}'>")
        html_parts.append("</head><body>")
        # Header with logo and title
        if "logo_path" in self.branding:
```

```
            html_parts.append(f"<div class='header'><img
src='{self.branding['logo_path']}' alt='Logo' style='max-height:60px;'/></
div>")
        html_parts.append(f"<h1>{title}</h1>")
        html_parts.append(f"<h2>Executive Summary</h2>")
        # Compute summary stats
        stats = self._compute_stats(issues)
        html_parts.append(f"<p>Total issues found: <b>{stats['total']}</b></
p>")
        html_parts.append(f"<p>High risk: <span class='high-
risk'>{stats['high']}</span>, "
                          f"Medium: <span class='medium-
risk'>{stats['medium']}</span>, "
                          f"Low: <span class='low-risk'>{stats['low']}</
span></p>")
        # (Potentially, include a traffic light chart or summary sentence)
        if stats['high'] > 0:
            html_parts.append(f"<p>Overall Risk Rating: <span class='high-
risk'>HIGH (Red)</span></p>")
        elif stats['medium'] > 0:
            html_parts.append(f"<p>Overall Risk Rating: <span class='medium-
risk'>MODERATE (Amber)</span></p>")
        else:
            html_parts.append(f"<p>Overall Risk Rating: <span class='low-
risk'>LOW (Green)</span></p>")
        html_parts.append("<hr/>")

        # Detailed findings by rule
        # Group issues by article or category
        issues_by_article = self._group_by_article(issues)
        for article, items in issues_by_article.items():
            html_parts.append(f"<h3>Findings for {article}</h3>")
            for issue in items:
                rule = issue.get("rule", "GDPR Requirement")
                verdict = issue.get("verdict", "")
                risk = issue.get("risk_level", "")
                rationale = issue.get("rationale", "")
                suggestions = issue.get("improvement_suggestions", [])
                snippet = issue.get("snippet", "")
                html_parts.append(f"<h4>{rule}: <span class='{risk.lower()}-
risk'>{verdict} ({risk})</span></h4>")
                if snippet:
                    html_parts.append(f"<blockquote>{snippet}</blockquote>")
                # Convert rationale and suggestions from markdown to HTML if
needed
                rationale_html = markdown.markdown(rationale)
                html_parts.append(f"<p>{rationale_html}</p>")
                if suggestions:
                    html_parts.append("<ul>")
                    for s in suggestions:
                        suggestion_html = markdown.markdown(s)
```

```
                        html_parts.append(f"<li>{suggestion_html}</li>")
                    html_parts.append("</ul>")
                # Direct quotes can be added as footnotes or inline citations
                # For simplicity, appending at the end of rationale for now
                quotes = issue.get("direct_quotes", [])
                for q in quotes:
                    html_parts.append(f"<p class='quote'>\"{q}\"</p>")
            html_parts.append("<hr/>")
        # Remediation section
        html_parts.append("<h2>Remediation Suggestions</h2><ul>")
        for issue in issues:
            if issue.get("verdict") != "Compliant":
                # Only suggest for problematic issues
                for s in issue.get("improvement_suggestions", []):
                    suggestion_html = markdown.markdown(s)
                    html_parts.append(f"<li>{suggestion_html} <em>(related
to {issue.get('rule')})</em></li>")
        html_parts.append("</ul>")

        # Footer with timestamp
        generated = datetime.datetime.now().strftime("%Y-%m-%d %H:%M")
        footer_text = self.branding.get("footer_text", f"Generated by
Blackletter Systems on {generated}")
        html_parts.append(f"<footer><p>{footer_text}</p></footer>")
        html_parts.append("</body></html>")
        return "".join(html_parts)

    def generate_pdf(self, issues: List[Dict]) -> str:
        """
        Build the HTML and generate a PDF file. Returns the file path of the
PDF.
        """
        html = self.build_html(issues)
        # If an external CSS file is provided, we load it
        stylesheets = []
        if self.css_path:
            stylesheets.append(CSS(filename=self.css_path))
        else:
            # default CSS is already inlined in HTML_parts in this case
            pass
        pdf_bytes = HTML(string=html).write_pdf(stylesheets=stylesheets)
        # Save PDF to a file (with a unique name)
        filename = f"gdpr_report_{uuid.uuid4().hex}.pdf"
        with open(filename, "wb") as f:
            f.write(pdf_bytes)
        return os.path.abspath(filename)

    def _compute_stats(self, issues: List[Dict]) -> Dict[str, int]:
        stats = {"total": len(issues), "high": 0, "medium": 0, "low": 0}
        for issue in issues:
            risk = (issue.get("risk_level") or "").lower()
```

```
            if risk.startswith("high"):
                stats["high"] += 1
            elif risk.startswith("med") or risk == "moderate":
                stats["medium"] += 1
            elif risk.startswith("low"):
                stats["low"] += 1
        return stats

    def _group_by_article(self, issues: List[Dict]) -> Dict[str, List[Dict]]:
        grouped = {}
        for issue in issues:
            # Assume issue['rule'] contains "Article X..." or similar
            article = "Other"
            rule = issue.get("rule", "")
            if "Article" in rule:
                idx = rule.index("Article")
                # e.g., "Article 5" take "Article 5" as key
                article = rule[idx:idx+9]  # crude way to get 'Article 5'
            grouped.setdefault(article, []).append(issue)
        return grouped

    def _default_css(self) -> str:
        # Basic CSS if no external stylesheet is used
        return """
        body { font-family: Arial, sans-serif; margin: 40px; }
        h1 { color: #2E4C6D; }
        h2 { color: #3A6EA5; margin-top: 1em; }
        h3 { color: #2E4C6D; margin-top: 1em; }
        .high-risk { color: red; font-weight: bold; }
        .medium-risk { color: orange; font-weight: bold; }
        .low-risk { color: green; font-weight: bold; }
        blockquote { background: #f9f9f9; border-left: 3px solid #ccc;
margin: 0.5em 0; padding: 0.5em; }
        footer { font-size: 0.8em; color: #888; text-align: center; margin-
top: 2em; }
        """
```

Let's break down what this `ReportBuilder` is doing: - It can either use a template file (with Jinja2) or build HTML manually. Using a Jinja2 template would allow a designer to edit the HTML structure easily, but here we also show a manual construction for clarity. - `_compute_stats` tallies how many issues of each risk level to use in the Executive Summary. - `_group_by_article` is a simple grouping mechanism to cluster issues by GDPR Article. In reality, our issue objects might already have an `article` field for grouping, which would be more robust. But this illustrates the idea. - The HTML includes: - A header with the firm logo and name. - Executive summary with total counts and colored risk counts. A simple logic to output an overall risk rating color. - Each article section lists the findings. We use headings for rule names and apply CSS classes to the verdict to color code (red for high risk issues, etc.). - The contract snippet is shown in a blockquote for context. - The rationale and suggestions are converted from Markdown to HTML. This is important because the LLM's rationale might include bullet points or other formatting that we want to preserve. Using `markdown.markdown()` helps format any Markdown syntax. (We ensure that our LLM outputs plain text or simple Markdown in rationale to benefit from this.) - Improvement suggestions are listed in an unordered list for each issue,

and again in the Remediation Suggestions section globally. - Direct quotes are included with a special style (e.g., `.quote` class) – one might alternatively footnote them with an index, but here we just display them in quotes for simplicity. - A horizontal line `<hr/>` is used to separate sections for clarity. - Finally, a footer with either custom text or a default text indicating when it was generated.

- `generate_pdf` calls `build_html` to get the HTML, then uses WeasyPrint to produce PDF bytes. It writes the PDF to a file, generating a unique filename (using `uuid` to avoid collisions). In a real deployment, instead of saving locally, this is where we might upload to S3:

  ```
  import boto3
  s3 = boto3.client('s3')
  s3.upload_fileobj(io.BytesIO(pdf_bytes), Bucket="reports-bucket",
  Key=filename, ExtraArgs={"ContentType": "application/pdf"})
  pdf_url = f"https://{s3_bucket}.s3.amazonaws.com/{filename}"
  return pdf_url
  ```

  But writing to local storage and returning the path can also suffice if the file is then served or emailed.

- The CSS defined covers basic styling, but in practice we'd likely have a more elaborate stylesheet (perhaps in a separate `.css` file that we include). This could define styles for the cover page, tables, etc., and handle page breaks (e.g., always page-break before a new article section, etc.).

Integration and Production Considerations

Both the LLM Judge and Report Builder modules are designed to integrate cleanly into the existing FastAPI + Celery pipeline: - In FastAPI, when a user submits a contract for review, an endpoint would create a job (entry in DB), enqueue tasks for clause extraction and evaluation. The LLM Judge might be invoked as part of those tasks (via Celery as discussed). Once all clause tasks are done, another task (or the main process) triggers the Report Builder. - The Report Builder could also be a Celery task: e.g., `generate_report_task(job_id)` that pulls the collected issues from the database and runs the builder. This task would then upload the PDF and mark the job as completed with a link to the report. - With Celery, we can easily run these tasks in the background and even schedule periodic tasks (though here it's likely on-demand per user request). - The code is organized into Python packages (e.g., `blackletter_llm_judge` and `blackletter_report_builder`). We make sure to include **docstrings** for every public class and method, explaining usage, and **type hints** for all parameters and return values, aiding in static analysis and developer understanding. This is critical for long-term maintainability when other developers at the firm integrate or modify the modules. - **Config for Playbook Overrides:** We mentioned that a YAML or JSON playbook could contain standard suggestions or even rule definitions. Both modules would allow injecting these configs. For instance, LLM Judge could accept a reference to a playbook so that if a rule is identified (say rule id "ART33"), it could attach a known citation or snippet of the law to the context. The Report Builder could use the playbook to fetch a standard recommendation text for a given issue code, rather than relying solely on the AI-generated suggestion. This approach brings together the best of automated analysis and human-curated guidance. - **Jurisdiction-aware filtering:** If the tool will be used beyond the UK (or if UK-specific deviations of GDPR apply), we can tag rules or issues with a jurisdiction and configure the system to include/exclude them. E.g., for UK, references to EU-only institutions might be skipped or replaced with UK equivalents. The Report Builder can filter out any issues not relevant to the chosen jurisdiction, ensuring the report is tailored to the user's regulatory context.

Conclusion

By implementing the **LLM Judge** and **Report Builder** as described, Blackletter Systems will have a robust AI-driven GDPR contract review capability. The LLM Judge provides a consistent, explainable evaluation of contract clauses against GDPR rules – complete with risk grading and improvement advice – while carefully controlling the LLM's output (using deterministic settings, templates, and guardrails to ensure accuracy and compliance with expected behavior). The Report Builder then takes these insights and creates a professional report, enhancing it with summaries, grouping, and visual cues like traffic-light risk indicators [10]. The final PDF/HTML report is suitable for sharing with stakeholders and forms a clear deliverable of the analysis.

Both modules are built with production deployment in mind: they are **modular, configurable, and integrable**. By using Python packages with clear APIs, they can be maintained or extended (for example, updating prompt templates for new types of checks, or adjusting report format per client feedback) without impacting other parts of the system. The use of Celery/NATS for background processing ensures scalability – multiple contracts can be analyzed in parallel, and large contracts can have their clauses processed concurrently to reduce turnaround time [8].

In summary, the approach provides a comprehensive and automated solution for GDPR contract compliance checking, combining the strengths of LLMs (natural language understanding and generation) with a structured, rule-based framework necessary for legal consistency and trustworthiness. All outputs are traceable and supported by source text citations, addressing the critical need for reliability in AI legal tools [1]. The final deliverable – the GDPR compliance report – empowers legal teams to quickly identify and address compliance gaps in their contracts, with the heavy lifting done by AI and the presentation tailored for human decision-makers.

Sources:

- OpenAI API documentation and community discussions on deterministic settings [3] – guiding the use of temperature=0 and low top_p to ensure repeatable outputs.
- Anthropic and Google model integration notes [5] – confirming adapter patterns for multiple LLM providers (OpenAI GPT-4, Claude, Google Gemini, etc.).
- AWS Prescriptive Guidance on LLM guardrails [6] – emphasizing prompt sanitization and input validation to secure AI systems.
- SO Development Blog on building trust in LLM answers [1] – highlighting the importance of source citations to avoid hallucinations and increase verifiability in domains like law.
- Confident AI on LLM-as-a-Judge [2] – discussing faithfulness of LLM outputs and sticking to facts rather than hallucinations.
- Celery official documentation [8] – describing Celery as a distributed task queue for handling background jobs (used for batch processing of clause evaluations).
- Binderr article on traffic light risk assessments [10] – explaining the Red/Amber/Green categorization of risk levels (used in our report's visual cues for risk).
- Dev.to tutorial on WeasyPrint [9] – demonstrating converting HTML/CSS to PDF in Python for report generation.
- Linode docs on Python-Markdown [13] – confirming the use of a library to convert Markdown content to HTML, allowing nicely formatted suggestions and rationale in the report.

---

[1] Building Trust in LLM Answers: Highlighting Source Texts in PDFs - SO Development
https://so-development.org/building-trust-in-llm-answers-highlighting-source-texts-in-pdfs/

2 LLM-as-a-Judge Simply Explained: The Complete Guide to Run LLM Evals at Scale - Confident AI
https://www.confident-ai.com/blog/why-llm-as-a-judge-is-the-best-llm-evaluation-method

3 Why does OpenAI API behave randomly with temperature 0 and ...
https://community.openai.com/t/why-does-openai-api-behave-randomly-with-temperature-0-and-top-p-0/934104

4 Why the API output is inconsistent even after the temperature is set to 0
https://community.openai.com/t/why-the-api-output-is-inconsistent-even-after-the-temperature-is-set-to-0/329541

5 agents-sdk-models - PyPI
https://pypi.org/project/agents-sdk-models/0.0.13/

6 FAQ - AWS Prescriptive Guidance
https://docs.aws.amazon.com/prescriptive-guidance/latest/llm-prompt-engineering-best-practices/faq.html

7 LLM Observability & Application Tracing (open source) - Langfuse
https://langfuse.com/docs/tracing

8 Celery - Distributed Task Queue — Celery 5.5.3 documentation
https://docs.celeryq.dev/en/stable/

9 11 12 Python PDF Generation from HTML with WeasyPrint - DEV Community
https://dev.to/bowmanjd/python-pdf-generation-from-html-with-weasyprint-538h

10 Why Your Traffic Light Risk Assessment Isn't Protecting You
https://binderr.com/resources/why-your-traffic-light-risk-assessments-arent-protecting-you-from-anything

13 Use Python-Markdown to Convert Markdown to HTML | Linode Docs
https://www.linode.com/docs/guides/how-to-use-python-markdown-to-convert-markdown-to-html/