ChatGPT

Blackletter GDPR Rules Engine Module

Overview

The Blackletter GDPR Rules Engine is a Python module designed to automatically check a contract (especially a **Data Processing Agreement** or similar contract) for the presence or absence of key GDPR-required clauses and identify ambiguous language. The engine performs the following steps:

- **Clause Segmentation:** Break the contract's raw text into numbered clauses using regex heuristics (e.g., detecting headings like "1.", "1.1", "Section 3.4.2", etc.) [1] .
- **Config Loading:** Load a GDPR rules checklist from a JSON file (with keywords and optional semantic aliases for each rule) and apply organization-specific overrides from a YAML "playbook" (enabling/disabling rules, adjusting severities, toggling vague term scan).
- **Context Expansion:** For each clause found, extend its context by ±2 surrounding clauses (paragraphs) to improve rule matching coverage.
- **Rule Matching:** Check each clause (with context) against the GDPR rule patterns using a hybrid approach – strict keyword matching and flexible alias/synonym matching – to find clauses that satisfy each rule or flag them as missing.
- **Vague Terms Detection:** Independently scan the entire document for **vague or ambiguous terms** (e.g., "reasonable efforts", "as soon as possible") that may require review [2] .
- **Issue Generation:** For each rule, produce a structured **Issue** object indicating whether the clause is found or missing, including details like clause reference, snippet, rationale, severity, and debug info. If a rule's clause is missing in the contract, a "missing clause" issue is emitted. Additionally, issues are generated for each occurrence of vague terms.
- **Integration:** The module is designed with a clear API (using classes and functions with type hints) so it can be called from a Celery task or a NATS message handler. Results can be returned (as Python objects or JSON) to an upstream **LLM Judge** or other pipeline components for further analysis or report generation.

We follow best practices in this implementation, including using Pydantic models for configuration and output schemas, extensive docstrings and comments for maintainability, and a modular structure (segmentation, loading, matching, etc.). This comprehensive answer includes code for the module, example usage with test inputs, and a brief developer guide.

Clause Segmentation

Contract text is first segmented into individual clauses based on common heading patterns. Many contracts use numbered headings and sub-headings (e.g., "1.", "1.1", "2.3.4") or prefixes like "Section" or "Article" [1] to delineate clauses. We use regular expressions to detect these patterns at the start of lines. For example, the regex will match:

- Pure numeric headings: `1.` , `1.1` , `1.1.1` , etc. (with or without a trailing period).
- Headings with words: `Section 3.4.2` , `Article 5` , `Clause 12.3` , etc., in a case-insensitive way.
- If no numbered headings are found, the entire text can be treated as a single clause.

The segmentation returns a list of clauses, each with an identifier (the clause number or title) and the clause text content. We ensure that the clause identifier (clause_path) does not include any trailing punctuation for consistency.

```python
import re

def segment_clauses(text: str):
    """
    Split the contract text into clauses based on heading patterns.
    Returns a list of clauses, where each clause is a dict with 'id' and
'text'.
    If no clause headings are found, returns one clause containing the whole
text.
    """
    # Regex to match clause headings (e.g., "1.", "1.1", "Section 2.3",
"Article IV", etc.)
    pattern = re.compile(
        r'^(?P<cid>(?:(?:Section|Article|Clause)\s+)?\d+(?:\.\d+)*\.?)',
        flags=re.IGNORECASE | re.MULTILINE
    )
    clauses = []
    matches = list(pattern.finditer(text))
    if not matches:
        # No headings found, treat entire text as one clause (id=None or
'All')
        clauses.append({"id": None, "text": text.strip()})
    else:
        for idx, match in enumerate(matches):
            cid = match.group('cid').rstrip('.')  # Remove trailing dot from
numbering
            start = match.start()
            # Determine end of this clause: start of next clause or end of
text
            end = matches[idx+1].start() if idx+1 < len(matches) else
len(text)
            clause_text = text[start:end].strip()
            clauses.append({"id": cid, "text": clause_text})
    return clauses
```

Explanation: The regex `^(?:(?:Section|Article|Clause)\s+)?\d+(?:\.\d+)*\.?` matches an optional "Section/Article/Clause" prefix, followed by one or more digits and optional sub-section digits (e.g., ".1", ".2.3"), and an optional trailing period. We iterate over all such matches in the text (using multiline mode to detect at each line) to find clause boundaries. Each clause's text runs from its start index to just before the next clause's start. The result is a list of clause dictionaries with `id` (the clause number or heading) and `text` (the full clause content). For example, a heading "1. Data Protection" will result in `id="1"` and the text including "1. Data Protection..." up to the next clause.

Configuration Models: GDPR Rules and Playbooks

The engine relies on two configuration files for flexibility:

1. **GDPR Rules Checklist (JSON):** Defines each GDPR rule to check for, including keywords that should appear in the contract clause and optional aliases or synonyms. Each rule also has a unique `id`, a human-readable description, and a default severity level. For instance, a rule might be defined for "Data Breach Notification" with primary keywords like `"72 hours", "breach"` and aliases like `"security incident"`.

2. **Organization Playbook (YAML):** Overrides or customizes the rules for a specific organization or context. In this YAML, you can enable/disable specific rules, adjust their severity, or add additional keywords/aliases. The playbook can also toggle on scanning for vague terms and even specify extra vague terms to look for. This allows tailoring the analysis per client or contract type (for example, disabling rules that aren't applicable, or marking some as more critical).

We use **Pydantic** data models to define the schema of these configurations and to load/validate them easily from JSON/YAML. Below are the data models:

```
from typing import List, Dict, Optional
from pydantic import BaseModel

class Rule(BaseModel):
    """GDPR Rule definition."""
    id: str
    description: str
    primary_keywords: List[str]
    aliases: List[str] = []
    severity: str = "Medium"

class RulesConfig(BaseModel):
    """Container for all GDPR rules loaded from JSON."""
    rules: List[Rule]

class RuleOverride(BaseModel):
    """Overrides for a single rule in a playbook."""
    enabled: Optional[bool] = None
# If set to False, this rule check is disabled
    severity: Optional[str] = None
# Override default severity (e.g., "Low", "High")
    add_keywords: List[str] = []           # Additional keywords to consider
for this rule
    add_aliases: List[str] = []            # Additional aliases/synonyms for
this rule

class Playbook(BaseModel):
    """Organization-specific playbook settings loaded from YAML."""
    organization: Optional[str] = None
    enable_vague_terms_scan: Optional[bool] = None
    # Rule overrides keyed by Rule ID
    rules: Dict[str, RuleOverride] = {}
```

How these models work: The `Rule` model holds the core definition of a GDPR checklist item (e.g., "Duty of Confidentiality clause must be present"), including which terms to look for. The `RulesConfig` might be loaded from a JSON file (list of rules). The `Playbook` model holds overrides; for example, if a certain company doesn't handle a particular scenario, they might disable that rule or lower its severity. Playbooks also have a flag to enable scanning for vague terms (which we'll discuss later). We merge the playbook settings with the default rules at runtime.

Example JSON (rules) and **YAML (playbook)** structure:

- **GDPR Rules JSON (simplified example):**

```
{
  "rules": [
    {
      "id": "GDPR_DATA_PROCESSING",
      "description": "Data Processing Clause (GDPR Article 28)",
      "primary_keywords": ["personal data", "processing", "data protection"],
      "aliases": ["GDPR", "Data Processing Agreement"],
      "severity": "High"
    },
    {
      "id": "GDPR_BREACH_NOTIFICATION",
      "description": "Personal Data Breach Notification (GDPR Article 33)",
      "primary_keywords": ["72 hours", "breach", "notify"],
      "aliases": ["data breach", "security incident"],
      "severity": "High"
    },
    ...
  ]
}
```

- **Playbook YAML (example overrides):**

```
organization: "ACME Corp"
enable_vague_terms_scan: true
rules:
  GDPR_BREACH_NOTIFICATION:
    severity: "Medium"    # Lower the severity for breach notification for
this org
  GDPR_DATA_EXPORT:
    enabled: false        # Disable a rule about data export if not
applicable
  # You can add extra keywords or aliases if needed, e.g.:
  GDPR_CONFIDENTIALITY:
    add_aliases: ["NDA", "non-disclosure"]
```

In code, loading these could be done via Pydantic's parsing, e.g.:

```
rules_conf = RulesConfig.parse_file("gdpr_rules.json")
playbook_conf = Playbook.parse_file("org_playbook.yaml")
```

(for brevity, file I/O is not shown here).

Vague Terms Detection

Apart from GDPR-specific clauses, it's a best practice to flag **vague or subjective terms** in contracts that could lead to ambiguity or disputes [2]. Terms like "reasonable", "appropriate", or "as soon as possible" are often considered *vague terms* because they lack precise definition and can be interpreted differently by each party. The rules engine can scan the entire contract for such terms if enabled in the playbook (`enable_vague_terms_scan: true`).

We maintain a default list of common vague terms, and this can be extended via the playbook if needed (for example, adding industry-specific ambiguous terms). The scan will find occurrences of these terms anywhere in the text (not just in numbered clauses) and create issues for review.

Default vague terms list (partial):

- *reasonable / reasonably* (e.g., "reasonable efforts")
- *undue delay* (e.g., "without undue delay")
- *promptly / as soon as possible / as soon as practicable*
- *best efforts* (or "commercially reasonable efforts")
- *material / substantial / substantially*
- *if appropriate / appropriate / adequate / sufficient / satisfactory*

These terms are flagged because they **could introduce uncertainty** in obligations or standards. For instance, the presence of **"reasonable efforts"** in a clause would be identified by our engine as a vague obligation that might need clarification [2] .

We will see in the code how the `enable_vague_terms_scan` flag triggers scanning of this list across the whole document and generates issues for each occurrence.

Rules Engine Core: Clause Matching and Context Expansion

With the clauses segmented and configurations loaded, the core engine will now evaluate each rule against the text. For each GDPR rule, it attempts to find a clause that matches the rule's criteria. Key points in this process:

- **Context Expansion:** Before matching, we build an extended context for each clause by including the text of up to 2 clauses before and after the clause. This helps catch scenarios where a rule's keywords might be split across neighboring clauses or a clause references something defined in a nearby clause. The extended context is just a concatenation of the neighboring clauses' text (without any special formatting, but we limit to ±2 to keep it relevant).

- **Keyword Matching:** We perform case-insensitive searches for the rule's **primary keywords** in the extended clause text. These primary keywords are usually the exact terms that must appear (or strongly indicate the clause fulfills the requirement). For example, for a **Security Measures clause**, primary keywords might include "technical and organizational measures" or "encryption".

- **Alias/Synonym Matching:** In addition to primary keywords, we check for any of the provided **aliases or semantic equivalents**. These could be abbreviations, alternate phrasings, or related terms. For instance, if a rule expects the presence of a Data Protection Officer, the text might use "DPO" – which would be listed as an alias.

- **Hybrid Approach:** A clause is considered a match for the rule if it contains *either* at least one primary keyword or an alias (or both). We use both strict matching (whole word/phrase matches via regex word boundaries) and can support partial/fuzzy matching if needed (e.g., via additional synonyms). In practice, our rules JSON defines the necessary synonyms explicitly, so a simple inclusion check suffices.

- **Multiple Occurrences:** If multiple clauses could match the same rule, the engine typically will flag the first or most relevant occurrence. (It is assumed each GDPR requirement is addressed in one main clause. If needed, the engine could be extended to handle multiple partial matches, but that is beyond scope here.)

- **Missing Clause:** If no clause in the contract contains any of the rule's keywords or synonyms, we flag that rule as missing from the contract. This is important because under GDPR Article 28, certain clauses are **mandatory** in contracts between controllers and processors [3] – if they're not found, it's a compliance gap.

Let's implement the main analysis function and supporting logic:

```
# Define a default list of vague terms (can be extended via playbook)
DEFAULT_VAGUE_TERMS = [
    "reasonable", "reasonably", "undue delay", "promptly",
    "as soon as possible", "as soon as practicable",
    "best efforts", "commercially reasonable",
    "material", "substantial", "substantially",
    "appropriate", "if appropriate", "adequate", "satisfactory", "sufficient"
]

def analyze_document(doc_id: str, text: str, rules_config: RulesConfig,
playbook: Optional[Playbook] = None) -> List['Issue']:
    """
    Analyze the contract text against GDPR rules and optional playbook
overrides.
    Returns a list of Issue objects for any found or missing clauses and
vague terms.
    """
    # 1. Segment the text into clauses
    clauses = segment_clauses(text)

    # 2. Prepare to collect issues
    issues: List[Issue] = []
    full_text_lower = text.lower()
    # 3. Apply playbook or use default if none
    if playbook is None:
        playbook = Playbook()  # empty playbook (no overrides)
    overrides = playbook.rules  # dict of rule overrides by id

    # 4. Iterate through each GDPR rule
    for rule in rules_config.rules:
        # Check if rule is disabled by playbook
        if rule.id in overrides and overrides[rule.id].enabled is False:
            continue  # skip this rule entirely


# Determine effective severity (use override if provided, else rule's
default)
        effective_sev = overrides.get(rule.id, RuleOverride()).severity or
rule.severity

        # Combine base keywords with any additional keywords/aliases from
playbook
        keywords = [kw.lower() for kw in rule.primary_keywords]
        aliases = [al.lower() for al in rule.aliases]
        if rule.id in overrides:
            keywords += [kw.lower() for kw in overrides[rule.id].add_keywords
or []]
            aliases  += [al.lower() for al in overrides[rule.id].add_aliases
or []]


# Compile regex patterns for each keyword and alias (word boundary to match
whole words/phrases)
        kw_patterns    = [re.compile(r'\b' + re.escape(kw)    + r'\b',
flags=re.IGNORECASE) for kw in keywords]
        alias_patterns = [re.compile(r'\b' + re.escape(al) + r'\b',
flags=re.IGNORECASE) for al in aliases]

        found = False
        found_clause_id = None
        snippet = ""
        matched_terms: List[str] = []

        # 5. Search through clauses with context expansion
        for i, clause in enumerate(clauses):

# Build extended context text for clause i (itself ±2 neighboring clauses)
            context_text_parts = []
            for j in range(i-2, i+3):
                if 0 <= j < len(clauses):
                    context_text_parts.append(clauses[j]["text"])
            context_text = " ".join(context_text_parts).lower()
            # Check for any keyword or alias matches in this context
            found_kw = any(p.search(context_text) for p in kw_patterns)
            found_al = any(p.search(context_text) for p in alias_patterns)
            if found_kw or found_al:
                found = True
                found_clause_id = clause["id"]
                # Capture snippet (e.g., first 200 characters of the clause
text for context)
                clause_text = clause["text"].strip()
                snippet = clause_text[:200] + ("..." if len(clause_text) >
200 else "")
                # Record which terms matched (for debugging)
                for p, term in zip(kw_patterns, keywords):
                    if p.search(context_text):
                        matched_terms.append(term)
                for p, term in zip(alias_patterns, aliases):
                    if p.search(context_text):
                        matched_terms.append(term)
                break  # no need to check further clauses for this rule
        # 6. Create Issue object for this rule
        if found:
            issue_status = "found"
            rationale_text = f"Clause covers requirement: {rule.description}"
        else:
            issue_status = "missing"
            rationale_text = f"No clause found covering: {rule.description}"
            found_clause_id = None
            snippet = ""
            matched_terms = []

        issue = Issue(
            id=f"{doc_id}_{rule.id}",
            doc_id=doc_id,
            rule_id=rule.id,
            clause_path=found_clause_id,
            snippet=snippet,
            citation=None,
# could be filled with law/regulation citation if available
            rationale=rationale_text,
            severity=effective_sev,
            status=issue_status,
            raw_matches={"keywords": matched_terms},
            rule_scores={
                "primary_found": any(term in keywords for term in
matched_terms),
                "alias_found": any(term in aliases for term in matched_terms)
            }
        )
        issues.append(issue)

    # 7. If enabled, scan for vague terms in the full text
    if playbook.enable_vague_terms_scan:
        # Combine default vague terms with any extra ones specified under a
special override key
        extra_vagues = []
        if "VAGUE_TERMS_EXTRA" in playbook.rules:
            extra_vagues = [t.lower() for t in
playbook.rules["VAGUE_TERMS_EXTRA"].add_keywords or []]
        vague_terms = [t.lower() for t in DEFAULT_VAGUE_TERMS] + extra_vagues
        for term in vague_terms:
            if term in full_text_lower:
                # Find an example snippet around the term for context
                idx = full_text_lower.index(term)
                snippet_start = max(0, idx - 40)
                snippet_end = min(len(text), idx + len(term) + 40)
                vague_snip = text[snippet_start:snippet_end].strip()
                # Create an issue for this vague term occurrence
                issue = Issue(
                    id=f"{doc_id}_VAGUE_{term}",
                    doc_id=doc_id,
                    rule_id="VAGUE_TERM",
                    clause_path=None,
                    snippet=(vague_snip + ("..." if snippet_end < len(text)
else "")),
                    citation=None,
                    rationale=f"Vague term '{term}' found, which may cause
ambiguity",
                    severity="Low",  # Vague terms are usually a low/medium
severity warning
                    status="review",  # status could indicate "review" needed
                    raw_matches={"term": [term]},
                    rule_scores={"vague_term_found": True}
                )
                issues.append(issue)
    return issues
```

In this `analyze_document` function, we:

1. Segment the text into clauses.
2. Loop over each rule in the `RulesConfig`:
3. Skip if disabled in playbook.
4. Merge any override keywords/aliases and determine severity.
5. Search through each clause (with context) for keywords.
6. Mark rule as found or missing accordingly.
7. Create an `Issue` data object for each rule (found or missing).
8. After rules, if vague term scanning is on, iterate through the list of vague terms and find any occurrences in the full text, generating an `Issue` for each found term.

The matching uses straightforward **case-insensitive substring matching** with regex word boundaries to avoid matching inside other words (for example, "processing" would match, but "processing" in "postprocessing" would not, due to word boundary).

The `matched_terms` collected are stored for debugging in `raw_matches`. The `rule_scores` field provides boolean flags indicating if any primary keyword was found and if any alias was found for that rule. This can help in debugging or for the LLM to know which part of the rule was satisfied.

Issue Object and Output Schema

We use another Pydantic model, `Issue`, to define the structure of each issue output. This ensures that the results are well-defined and easy to serialize (e.g., to JSON for the next stage in the pipeline). An **Issue** contains:

- `id`: Unique issue identifier (we combine the document ID and rule ID for uniqueness).
- `doc_id`: The document identifier (passed into the function, representing the contract being analyzed).
- `rule_id`: Which rule this issue pertains to (e.g., "GDPR_BREACH_NOTIFICATION" or a special "VAGUE_TERM").
- `clause_path`: The clause identifier in the contract (e.g., "Section 5.2" or "4.1") where the match was found. If the issue is a missing clause or a vague term not tied to a specific numbered clause, this may be `None` or `null`.
- `snippet`: A short excerpt of the contract text illustrating the context. For found clauses, this will be the beginning of the clause that triggered the match. For vague terms, it will show the term in context. This helps reviewers quickly see the relevant text.
- `citation`: (Optional) A citation or reference. This could be used to point to an external source or law (for example, the GDPR Article number). In our implementation we left it `None`, but one could populate this with something like `"GDPR Art. 33"` for a breach notification rule.
- `rationale`: A brief explanation or rationale for the issue. For found clauses, it might say the clause covers the requirement. For missing clauses, it explicitly notes no clause was found for that requirement. For vague terms, it notes the term may cause ambiguity.
- `severity`: The severity level of the issue (e.g., "High", "Medium", "Low"). This comes from the rule definition and can be overridden by the playbook. Missing critical clauses would likely be "High" severity, whereas vague terms might be "Low" or "Medium".
- `status`: The status or category of the issue. We use `"found"` for a present clause, `"missing"` for an absent clause that should exist, and `"review"` for vague terms (indicating it's something to review rather than a definite compliance gap).
- `raw_matches`: Debug info about what terms were matched. For rule-based issues, this includes a list of keywords from the rule that were found in text (could include both primary and alias terms). For vague term issues, it includes the term itself.
- `rule_scores`: Additional debug booleans – e.g., `primary_found` and `alias_found` for rules. This can help an auditor or developer see *how* the rule was satisfied (did it match a primary keyword, an alias, or both?). For vague terms, we simply set a `vague_term_found` flag.

Below is the `Issue` model definition:

```python
class Issue(BaseModel):
    """Result of a rule check or vague term detection."""
    id: str
    doc_id: str
    rule_id: str
    clause_path: Optional[str]
    snippet: str
    citation: Optional[str]
    rationale: str
    severity: str
    status: str
    raw_matches: Dict[str, List[str]]
    rule_scores: Dict[str, bool]
```

This structured output makes it easy for the next steps in the pipeline (for example, an LLM-based report generator) to iterate through issues and formulate a summary. If needed, it can be exported to JSON via `Issue.json()` or a list of issues to `[issue.dict() for issue in issues]`.

Missing Clause Handling and GDPR Compliance Gaps

One of the core purposes of this engine is to detect when a required GDPR clause is missing from the contract. Under GDPR (Article 28(3) and related provisions), a contract between a data controller and processor **must** include certain clauses (such as on instructions, confidentiality, security, sub-processing, data subject rights, etc.) [3] . Our rules checklist is designed to cover these required terms. If the engine does not find a clause corresponding to one of these rules, it emits an issue with `status="missing"`.

For example, if a Data Processing Agreement lacks a clause about **breach notification** (which is a GDPR requirement under Articles 33/34), the engine would output a missing issue like:

```
{
  "id": "DOC123_GDPR_BREACH_NOTIFICATION",
  "doc_id": "DOC123",
  "rule_id": "GDPR_BREACH_NOTIFICATION",
  "clause_path": null,
  "snippet": "",
  "citation": null,
  "rationale": "No clause found covering: Personal Data Breach Notification
(GDPR Article 33)",
  "severity": "High",
  "status": "missing",
  "raw_matches": {"keywords": []},
  "rule_scores": {"primary_found": false, "alias_found": false}
}
```

This highlights a compliance gap that should be remedied. Each rule in the JSON can be tied to a GDPR article or principle, so in practice the `citation` or `description` field would reference that (as shown in the rationale above).

By contrast, if a rule is found, the issue will have `status="found"`, a clause reference, and a snippet. This helps verify that the contract has the necessary language. For instance, a found issue might look like:

```
{
  "id": "DOC123_GDPR_DATA_PROCESSING",
  "doc_id": "DOC123",
  "rule_id": "GDPR_DATA_PROCESSING",
  "clause_path": "1",
  "snippet":
"1. Data Protection\nThe parties agree to comply with applicable data
protection law including the GDPR. The Processor shall implement appropriate
technical and ...",
  "citation": null,
  "rationale": "Clause covers requirement: Data Processing Clause (GDPR
Article 28)",
  "severity": "High",
  "status": "found",
  "raw_matches": {"keywords": ["personal data", "data protection", "gdpr"]},
  "rule_scores": {"primary_found": true, "alias_found": true}
}
```

In this example, the engine found clause "1" (Data Protection) containing terms "personal data", "data protection", and "GDPR", satisfying the rule. The raw matches show which keywords triggered the detection (both primary and alias terms were found).

Usage Example and Testing

To illustrate how the module works, let's walk through a quick example. Imagine we have a sample contract text and we want to run the GDPR rules engine on it:

Sample Contract Excerpt:

```
1. Data Protection
The parties agree to comply with all applicable data protection laws
including the GDPR. The Processor shall implement appropriate technical and
organizational measures to protect personal data.

2. Confidentiality
Each party shall use reasonable efforts to maintain the confidentiality of
the other party's information.
```

This text has a clause 1 covering data protection (with keywords like "data protection", "GDPR", "personal data") and a clause 2 about confidentiality which contains the vague term "reasonable efforts". Suppose our rules JSON includes two rules: one for a Data Protection clause (Article 28) and one for Breach Notification (Article 33). The breach notification clause is absent in the text. We enable vague term scanning in the playbook.

Using the module would involve loading the configs and calling `analyze_document`:

```
# Load rules and playbook (this would normally be from files)
rules_conf = RulesConfig.parse_obj(rules_json_dict)
playbook_conf = Playbook.parse_obj(playbook_yaml_dict)

# Analyze the document text
```

```python
issues = analyze_document(doc_id="DOC123", text=contract_text,
                          rules_config=rules_conf, playbook=playbook_conf)

# Output results (for demonstration, just printing each issue)
for issue in issues:
    print(issue.json(indent=2))
```

After running this, the engine would produce a list of Issue objects. For our sample, the output might be:

- **GDPR_DATA_PROCESSING**: Found in clause 1 (severity High).
- **GDPR_BREACH_NOTIFICATION**: Missing (severity overridden to Medium in playbook).
- **VAGUE_TERM** ("reasonable"): Found in clause 2, needing review (severity Low).

Each issue is output as a JSON (via Pydantic) for clarity. This confirms that clause 1 satisfies the data protection rule, clause 2 contains a vague term "reasonable", and the breach notification clause was not found, which is flagged as a gap.

Developer Notes and Integration Guide

Project Structure: This module can be organized into files for clarity, for example: - `models.py` containing Pydantic models (`Rule`, `Playbook`, `Issue`, etc.). - `utils.py` containing helper functions like `segment_clauses` and maybe text cleaning or fuzzy matching utilities. - `engine.py` containing the main logic (`analyze_document` function). - Optionally separate modules for loading files (if needed) or fuzzy matching extensions.

For simplicity, we've presented the module in a single cohesive script above, but it can be refactored into a package structure as needed.

Dependencies: The code uses the Pydantic library for data models, so ensure `pydantic` is installed. Standard libraries like `re` are used for regex. No other external libraries are required unless you want to integrate more advanced NLP for semantic matching.

Testing: We recommend writing unit tests for: - Clause segmentation (ensure that various numbering formats are correctly identified). - Rule matching (create dummy texts that should trigger or not trigger certain rules). - Playbook overrides (test that disabling a rule or adding a keyword works as expected). - Vague term detection (text containing known vague terms should produce issues).

Some basic test cases were illustrated in the usage example above. The expected outputs can be asserted in tests by comparing the Issue objects or their JSON representation.

Integration with Celery/NATS: The function `analyze_document` is designed to be easily invoked from a Celery task or a message queue consumer: - In a Celery task, you might do something like: `issues = analyze_document(doc_id, text, rules_conf, playbook_conf)` and then return or store the `issues` (Pydantic models can be converted to dict for sending as JSON). - In a NATS message handler, similar usage applies. Make sure to serialize the output (e.g., using `jsonable_encoder` from Pydantic or simply converting to dict) before sending over the wire.

The output is structured and can be fed to an LLM-based "Judge" or reporter. For instance, an LLM could iterate through issues and generate a human-readable report or suggestions (like "Your contract is missing a Breach Notification clause required by GDPR, which is high severity.").

Developer Onboarding: The code is thoroughly commented and uses descriptive names, making it easier to trace. New developers should start by understanding the `RulesConfig` and `Playbook` structures, then how `analyze_document` processes the text. The use of **Pydantic** ensures that inputs/outputs adhere to the expected schema, reducing errors. If modifications are needed (e.g., adding a new rule or a new type of check), one can update the JSON/YAML and adjust logic accordingly.

Finally, remember that this engine is a starting point. It can detect presence/absence of clauses and flag vague language, which greatly speeds up compliance reviews. However, it might not fully understand context or legal nuance – that's where either human expertise or advanced AI analysis (the LLM Judge) comes in to interpret the findings and provide guidance.

---

1 Contract articles, sections and contract clause numbering - Weagree
https://weagree.com/clm/contracts/contract-structure-and-presentation/articles-sections-clause-numbering/

2 A Contract Review Checklist to Protect from Legal Risks
https://www.hyperstart.com/blog/contract-review-checklist/

3 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/