CHATBOT PROMPTS

**1. Document Analysis & Compliance Matching**

* *"Analyze this legal document and extract all compliance-related clauses."*
* *"Compare this contract against [Regulation X] and highlight non-compliant sections."*
* *"Does this policy document adhere to GDPR requirements? List discrepancies."*
* *"Extract all obligations and deadlines from this compliance document."*
* *"Identify any conflicts between this internal policy and [Industry Standard Y]."*

**2. Query Handling & FAQ Responses**

* *"What are the key compliance requirements for [Financial Sector] in [Country/Region]?"*
* *"Explain the difference between SOC 2 and ISO 27001 compliance frameworks."*
* *"How do I ensure my document meets HIPAA compliance standards?"*
* *"List the mandatory clauses for a vendor agreement under [Regulation Z]."*
* *"What penalties apply for non-compliance with [Law/Standard ABC]?"*

**3. Automated Compliance Checks & Risk Assessment**

* *"Scan this document for potential compliance risks and suggest mitigations."*
* *"Generate a compliance checklist for a new [Healthcare/FinTech] project."*
* *"Assess this procurement policy for anti-bribery (FCPA) compliance."*
* *"Flag any ambiguous language in this contract that could lead to compliance issues."*
* *"Predict potential compliance risks if we expand operations to [New Country]."*

**4. Workflow & Process Automation**

* *"Draft a compliance approval workflow for new internal policies."*
* *"Automate tracking of regulatory updates affecting our industry."*
* *"Suggest an audit trail format for compliance document revisions."*
* *"Create a template for a compliance gap analysis report."*
* *"How can we integrate this chatbot with our existing GRC (Governance, Risk, Compliance) software?"*

**5. Testing & Improving the Chatbot**

* *"If I ask about [specific compliance topic], how would you respond?"*
* *"What’s your confidence level in this compliance recommendation?"*
* *"Simulate a conversation where a user asks about PCI DSS compliance."*
* *"Identify areas where your compliance knowledge may be lacking."*
* *"Suggest improvements to make your compliance matching more accurate."*

**Bonus: Edge Cases & Troubleshooting**

* *"What do you do if a document’s compliance status is unclear?"*
* *"How would you handle conflicting regulations in different jurisdictions?"*
* *"Can you explain a complex compliance rule in simple terms for non-experts?"*

**🔍 Deep Compliance Analysis & Cross-Referencing**

1. **"Take [Document A] and [Regulation B], then:**
   * List sections where Document A **exceeds** Regulation B’s requirements.
   * Identify where it **falls short**.
   * For each shortcoming, reference [Industry Best Practice C] and suggest revisions."\*\*
2. **"Analyze these three policies ([X], [Y], [Z]) for conflicts with [Law L]:**
   * Rank them by compliance risk (high/medium/low).
   * For the highest-risk policy, draft an amendment that resolves the conflict."\*\*
3. **"Compare [Contract Clause 1] and [Court Ruling 2]:**
   * Does the clause align with the ruling’s interpretation?
   * If not, rewrite it to avoid legal risk, citing [Precedent Case 3]."\*\*

**🔄 Self-Referential & Improvement-Focused**

1. **"Review my last five compliance queries. Based on my questions:**
   * What gaps do I likely have in my compliance knowledge?
   * Recommend 3 key documents/regulations I should study next."\*\*
2. **"Here’s a compliance report I generated earlier ([Attached]).**
   * Critique its strengths/weaknesses against [Standard S].
   * Rewrite the weakest section to meet [Standard S] + add audit-proof language."\*\*
3. **"I’m working on [Project P] under [Framework F]. You previously suggested [Action A]. Now:**
   * Has [Recent Amendment R] affected its validity?
   * If yes, propose an updated action plan with timeline."\*\*

**⚡ Multi-Step Hypotheticals (Interdependent Logic)**

1. **"Assume my company violates [Regulation R] due to [Scenario S]:**
   * What immediate corrective actions are needed?
   * Which internal teams must be involved?
   * Draft a disclosure statement referencing [Guideline G]."\*\*
2. **"If I merge [Policy X] (aligned with GDPR) and [Policy Y] (aligned with CCPA):**
   * Identify 3 conflicts in data retention clauses.
   * Create a unified policy that satisfies both, prioritizing stricter clauses."\*\*
3. **"Hypothetical: A vendor’s [Contract C] lacks [Clause D] required by [Law L]:**
   * What’s the short-term workaround?
   * Long-term, how should we update our vendor onboarding process?"\*\*

**📌 Meta-Prompts (For Chatbot’s Own Improvement)**

1. **"You just answered my query about [Topic T]. Now:**
   * What external sources (laws/cases) did you rely on?
   * Rate your confidence (1-10) in this answer.
   * If it’s below 8, what additional data would raise it?"\*\*
2. **"Simulate a debate between two compliance experts:**
   * One argues [Position A] based [Document X].
   * The other argues [Position B] based [Document Y].
   * Synthesize a compromise using [Regulation Z]."\*\*
3. **"Track all compliance questions I’ve asked this month:**
   * Generate a heatmap of my high-risk focus areas.
   * Recommend a training plan to address blind spots."\*\*

**🌍 Cross-Border & Multijurisdictional Compliance**

1. **"Compare Basel III, Dodd-Frank, and MiFID II liquidity requirements:**
   * Highlight 3 key conflicts for a bank operating in the US and EU.
   * Draft a unified liquidity policy that satisfies the strictest clauses from each."\*\*
2. **"A transaction involves USA (BSA/AML), EU (AMLD6), and Singapore (MAS 626):**
   * Identify redundant KYC checks across these regimes.
   * Propose an optimized workflow to avoid duplication."\*\*
3. **"Our bank’s GDPR data retention policy conflicts with CCPA’s ‘right to delete’:**
   * Design a tiered deletion system (e.g., EU vs. California customers).
   * Reference [ECJ Case X] and [CPRA Guidelines Y] in your solution."\*\*

**⚠️ High-Stakes Risk Assessment & Mitigation**

1. **"Simulate a FATF audit on our AML program:**
   * List 5 ‘red flag’ gaps using the FATF 40 Recommendations.
   * For each gap, suggest a corrective action with owner (e.g., Legal/IT)."\*\*
2. **"An employee bypassed SWIFT screening controls (like the Danske Bank case):**
   * What real-time monitoring rules could prevent this?
   * How would you update our SAR (Suspicious Activity Report) template?"\*\*
3. **"Stress-test our sanctions screening system against OFAC’s 50% rule:**
   * Generate a risk score (1-10) for false negatives.
   * Recommend AI/ML models to reduce it (e.g., graph analytics for shell companies)."\*\*

**📑 Document Intelligence & Audit Proofing**

1. **"Extract all ‘material adverse change’ clauses from 10 loan agreements:**
   * Cluster them by jurisdiction (US/UK/HK).
   * Flag any outliers against ISDA 2020 definitions."\*\*
2. **"Analyze our trade surveillance logs vs. MAR (Market Abuse Regulation):**
   * Identify 3 patterns resembling insider trading (e.g., pre-announcement spikes).
   * Auto-generate an audit trail for regulators."\*\*
3. **"Our LIBOR transition documentation references SOFR and SONIA:**
   * Verify if fallback language aligns with ARRC/ISDA standards.
   * Highlight any ‘zombie LIBOR’ exposure in legacy contracts."\*\*

**🤖 AI Chatbot Training & Improvement**

1. **"Based on my last 10 queries, I focus on AML and cross-border tax (FATCA/CRS):**
   * What 3 lesser-known regulations (e.g., UAE AML Decree 20) should I prioritize?
   * Suggest a training module (with quiz) to cover them."\*\*
2. **"You previously recommended ‘behavioral analytics’ for AML. Now:**
   * Compare vendors like Quantexa vs. Nice Actimize for our transaction volume.
   * Draft an RFP section on ‘AI model explainability’ for regulators."\*\*
3. **"Rate your confidence (1-10) in your last 5 answers on MiCAR (EU Crypto Rules):**
   * For answers below 8, pull the latest EBA guidelines to self-correct."\*\*

**🔥 Worst-Case Scenarios & Crisis Response**

1. **"We face a ‘Deutsche Bank’-style mirror trading allegation:**
   * Immediate steps to isolate compliance team communications (attorney-client privilege).
   * Long-term, how to restructure our FX trading oversight?"\*\*
2. **"A whistleblower reports nostro account misuse (like Banco Espirito Santo):**
   * Generate a 48-hour containment plan (legal + PR).
   * Update our FRB Resolution Plan ‘living will’ to address this."\*\*
3. **"Hypothetical: Our AI model falsely flags 40% of Muslim names for sanctions:**
   * How to retrain it using Fair Lending principles?
   * Draft an apology template for impacted customers."\*\*

**📌 Core Principle:**

**"Never answer without citing a specific section/clause from my uploaded documents. If unsure, say: ‘Reference missing in [Document X] for [Topic Y]—upload the relevant text.’"**

**🔍 Document-Specific Compliance Checks**

1. **"Scan [Uploaded AML Policy] against [Uploaded FATF 40 Recommendations PDF]:**
   * List every FATF rule **not explicitly addressed** in Section 3 of our policy.
   * For missing rules, paste the exact FATF clause and draft a policy addition."\*\*
2. **"Compare [Uploaded Loan Agreement]’s ‘Default Events’ (Clause 5.2) to [Uploaded ISDA Master Agreement]’s ‘Termination Events’:**
   * Create a table showing gaps (e.g., ‘ISDA includes “Credit Support Default” but Loan Agreement doesn’t’).
   * Rewrite Clause 5.2 to mirror ISDA’s stricter terms."\*\*
3. **"In [Uploaded GDPR Data Retention Schedule], does ‘Section 4: Erasure Protocol’ align with [Uploaded ECJ Case C-131/12 (Google Spain) ruling]?**
   * Quote the case’s ‘right to delist’ paragraph and flag conflicts with our erasure timelines."\*\*

**⚡ Dynamic Cross-Referencing**

1. **"Cross-check [Uploaded Vendor Contract]’s ‘Data Protection’ clause (Section 8) with:**
   * [Uploaded Bank’s Internal Data Privacy Policy] Section 2.3
   * [Uploaded CCPA Text] § 1798.120(a)
   * **Output:** A redlined clause showing where the contract fails to meet internal/external rules."\*\*
2. **"Merge [Uploaded FRB SR 08-8] and [Uploaded PRA SS2/21] on operational resilience:**
   * Extract **3 overlapping requirements** (e.g., ‘incident reporting within 72 hours’).
   * For conflicts, prioritize the stricter rule and justify."\*\*
3. **"Does [Uploaded Internal Trade Surveillance Log] show patterns violating [Uploaded MAR Article 12]’s ‘manipulation of benchmarks’ definition?**
   * Isolate logs matching MAR’s examples (e.g., ‘layering’ or ‘spoofing’).
   * Output as a pseudo-SAR (Suspicious Activity Report) citing MAR paragraphs."\*\*

**🛠️ Gap Analysis & Auto-Correction**

1. **"Analyze [Uploaded Sanctions Screening SOP] vs. [Uploaded OFAC 50% Rule Guidance]:**
   * Does our SOP’s ‘ownership calculation’ method match OFAC’s examples? If not, rewrite it verbatim."\*\*
2. **"In [Uploaded Whistleblower Policy], is ‘anonymity’ guaranteed per [Uploaded SEC Rule 21F-17(a)]?**
   * Highlight where our policy uses weaker language (e.g., ‘reasonable efforts’ vs. ‘ensure’)."\*\*
3. **"Extract all ‘material adverse change’ clauses from [Uploaded 10 Loan Agreements]:**
   * Cluster definitions by jurisdiction (US/UK/HK) using [Uploaded ISDA 2020 Definitions] as the baseline.
   * Flag any agreement with a definition **broader** than ISDA’s."\*\*

**🚨 Crisis Mode (Strictly Document-Led)**

1. **"A breach occurred. Validate our response against [Uploaded Incident Response Plan] and [Uploaded CBEST Framework]:**
   * Does Section 4.1 (‘Escalation Paths’) align with CBEST’s ‘Tier 3 Cyber Attack’ protocol?
   * If not, paste CBEST’s required steps missing in our plan."\*\*
2. **"Using [Uploaded Board Minutes] and [Uploaded SR 13-19], prove we’ve discussed interest rate risk in the past 6 months:**
   * Quote exact minutes where ‘IRRBB’ was mentioned. If absent, list SR 13-19’s required discussion topics."\*\*
3. **"Does [Uploaded Employee Trading Policy]’s ‘Pre-Clearance’ rule match [Uploaded FCA SYSC 10.2]’s ‘personal account dealing’ requirements?**
   * Tabulate differences (e.g., ‘FCA requires quarterly disclosures; our policy says annual’)."\*\*

**🤖 Chatbot Self-Audit (No Generic Answers)**

1. **"From my last 10 uploads, which 3 documents have the most outdated references (e.g., cite repealed laws)?**
   * List the outdated clauses and their current versions (e.g., ‘MiFID II replaced MiFID in 2018’)."\*\*
2. **"In [Uploaded Chatbot Training Manual], does ‘Section 7: Legal Citations’ align with [Uploaded ABA Model Rules on AI Use]?**
   * Rewrite any non-compliant guidance (e.g., ‘always attribute to primary law, not summaries’)."\*\*
3. **"Train yourself: Compare [Uploaded ERM Framework] and [Uploaded COSO ERM Guide].**
   * Generate a quiz question testing the difference between ‘risk appetite’ definitions in each."\*\*

**🔥 Nightmare Scenarios (State-Sponsored ML/Terror Finance)**

1. **"Cross-reference [Uploaded SWIFT Transaction Logs] with [Uploaded OFAC SDN List] and [Uploaded UNSC 1267 List]:**
   * Flag any payments to **Entity X** (a front company linked to [Uploaded FBI FinCEN Advisory 2023-456]).
   * **Strict Rule:** Only tag matches if the beneficiary name **exactly aligns** with definitions in [Uploaded Glossary.docx]’s ‘Shell Company Naming Conventions’ section."
2. **"Analyze [Uploaded Corporate Client KYC File] for signs of North Korean ransomware cashouts per [Uploaded FATF Illicit Typologies Report 2024]:**
   * Is there a mismatch between declared revenue and [Uploaded Blockchain Explorer Data]? Cite **exact page numbers** from the FATF report.
   * **Glossary Lock:** Only classify as ‘high risk’ if the client matches [Uploaded Glossary.docx]’s ‘DPRK-Linked Entity’ criteria (e.g., ‘uses Hong Kong intermediaries’)."
3. **"A Venezuelan state oil company ([Uploaded Client Y]) requests a USD transfer to Malta. Validate against:**
   * [Uploaded US EO 13808] (Venezuela Sanctions)
   * [Uploaded ECB AML/CFT Guidelines] §17.2 (‘high-risk jurisdictions’)
   * **Output:** A ‘Block/Release’ decision matrix **quoting exact clauses** from both docs. If terms conflict, follow [Uploaded Internal Escalation Protocol.docx] Section 9 (‘Sanctions Tiebreaker Rule’)."

**⛓️ Absolute Document Binding (No AI "Creativity")**

1. **"Define ‘Politically Exposed Person’ only using [Uploaded Glossary.docx]’s definition (v2.3). Now:**
   * Screen [Uploaded Board Member List] against this definition.
   * **Reject** any matches not meeting the **exact wording** (e.g., ‘immediate family’ ≠ ‘cousins’ unless glossary says so)."
2. **"Calculate ‘suspicious transaction threshold’ only using the formula in [Uploaded AML Policy.pdf] Appendix C:**
   * Apply it to [Uploaded Crypto Transaction Batch].
   * **Invalidate** any alerts where the math doesn’t match **down to 2 decimal places**."
3. **"Draft a SAR (Suspicious Activity Report) for [Uploaded Unusual Wire Activity.csv]:**
   * **Every field** must pull from [Uploaded FinCEN SAR Field Guide 2024].
   * If a field (e.g., ‘suspicion type’) isn’t explicitly defined there, output: ‘[FIELD] definition missing in [Uploaded FinCEN Guide]. Upload exact wording.’"

**💀 Crisis Escalation (Legal/Reputational Doomsday)**

1. **"We just processed a $200M ‘mirror trade’ resembling the Deutsche Bank scandal ([Uploaded Court Filing ABC-123]):**
   * Immediately extract **all employees** named in [Uploaded Internal Access Logs] who touched this trade.
   * Cross-check against [Uploaded HR Records] for ‘disciplinary history’ per [Uploaded FRB Enforcement Action 2022-5]’s ‘individual accountability’ clause."
2. **"A leak reveals nostro accounts were used to evade Iran sanctions ([Uploaded Reuters Article 2024]):**
   * Freeze all accounts matching the **transaction patterns** in [Uploaded Internal Investigation Report 789].
   * **Redact** all data not explicitly required by [Uploaded DOJ Voluntary Disclosure Agreement.docx] Section 4(b)."
3. \*\*"Our AI model **biasedly flagged 90% of Yemeni remittances** as ‘high risk’:
   * Audit the model’s training data against [Uploaded Fair Lending Act] and [Uploaded OCC Bulletin 2023-15].
   * **Revert all false positives** and output an apology template **quoting** [Uploaded PR Crisis Handbook]’s ‘Regulatory Mea Culpa’ language."

**🛑 Zero-Tolerance Output Rules**

1. **"From now on, ALL outputs must:**
   * Cite **page/line numbers** from uploaded docs (e.g., ‘[Uploaded BSA.pdf] p.45, line 12’).
   * **Reject** requests lacking uploads (e.g., ‘Define OFAC 50% rule’ → ‘Error: Upload OFAC guidance first’).
   * Use **exact synonyms** from [Uploaded Glossary.docx] (e.g., ‘terrorist’ ≠ ‘militant’ unless glossary equates them)."
2. **"Validate this response against [Uploaded Compliance Bot SOP] Section 3.2 (‘No Interpretation Allowed’):**
   * If I ask ‘Is this activity suspicious?’ but haven’t uploaded [Uploaded SAR Decision Matrix], output: ‘Denied: Missing [Uploaded SAR Matrix]. Rule: SOP 3.2.’"
3. **"Simulate a regulator cross-examining the bot ([Uploaded SEC Deposition Transcript 2023]):**
   * For every answer, prove its origin in uploaded docs.
   * If challenged (e.g., ‘How did you calculate this risk score?’), output: ‘Source: [Uploaded Model Card.docx] Equation 2.1’."

## 🔍 SECTION A: Document Comprehension (Upload-Based Context)

### Goal: Test if the model ****understands and refers to uploaded control and regulatory clauses****.

1. What are the top 3 compliance risks found in my uploaded controls?
2. Which clauses from the control repository are weakly mapped?
3. Identify control clauses that don't match any regulatory clause.
4. Are there any GDPR-specific gaps in my policy?
5. Explain clause Control-5 in simpler terms.
6. What does clause ISO27001-Clause-8 refer to in layman’s language?
7. Which uploaded regulation talks about data retention policies?
8. How many unmatched clauses are there in total?
9. List all matched clauses and their respective regulation names.
10. Which controls failed to comply with SEBI’s risk reporting requirements?

## 🧠 SECTION B: Deep Reasoning & Semantic Gap Analysis

### Goal: Evaluate ****semantic comparison + AI’s reasoning quality****.

1. Why is Control-7 considered only a partial match?
2. What is the semantic difference between Control-3 and the matched GDPR clause?
3. Give 3 reasons why Control-10 does not fully align with RBI’s cyber regulation.
4. Suggest a better rewrite for Control-11 to match the corresponding regulatory clause.
5. What overlap exists between Control-4 and the regulatory clause it matched?
6. Can you highlight the missing terms in Control-2 based on its regulatory match?
7. Which clause would you recommend rewriting due to high non-compliance risk?
8. How does Control-6 fall short of DPDP expectations?
9. What keywords are missing from the control related to user consent?
10. Rewrite Control-8 so it better aligns with ISO27001 clause 5.3.

## 🧾 SECTION C: Compliance Gap Identification

### Goal: Confirm if AI accurately pinpoints ****regulatory gaps**** in policies.

1. Which key GDPR articles are missing from the control repository?
2. Are there any gaps in access control procedures as per ISO27001?
3. Do my controls meet RBI’s incident response requirements?
4. What are the top 5 semantic gaps across all matched clauses?
5. List the regulatory requirements not covered by any control.
6. What’s the biggest gap found in data encryption policies?
7. Identify any mismatched clauses where AI is unsure of compliance.
8. What clauses in my controls mention audit logs but miss regulatory mandates?
9. Is there a mismatch between my data localization clause and SEBI’s norms?
10. Can you spot any missing references to data subject rights under GDPR?

## ⚠️ SECTION D: Risk, Fine & Compliance Suggestions

### Goal: Test AI’s understanding of ****risk language, fines, and regulatory penalties****.

1. What’s the estimated risk level for Control-9?
2. Suggest the fine level for a missing clause related to GDPR’s consent article.
3. How risky is it to not comply with the ISO27001 control for third-party access?
4. Is the clause about breach notification aligned with GDPR timelines?
5. What non-compliance impact would occur if we missed RBI’s KYC directive?
6. What fine bracket (low, medium, high) applies to Control-12 and why?
7. Suggest a better compliance approach for our cloud storage control.
8. What penalties apply if we fail to implement a privacy policy under DPDP?
9. How can I reduce the fine risk associated with Control-14?
10. Suggest one practical rewrite that improves compliance without increasing complexity.

## 🧑‍💼 SECTION E: Regulatory Reasoning (Multi-Regulation Match)

### Goal: Test ****AI’s ability to map 1 control to multiple regulations****.

1. Which regulation does Control-13 match most strongly with?
2. Can one clause map to both GDPR and DPDP? Show an example.
3. Which controls cover both RBI and SEBI expectations?
4. Map Control-15 to all relevant regulations.
5. Identify the most universal clause across all regulatory uploads.
6. Does any control fulfill both ISO27001 and GDPR?
7. Show overlap between one control clause and two regulatory requirements.
8. Is there a control clause that’s over-compliant across multiple jurisdictions?
9. Show me a clause that is under-compliant in one region but compliant in another.
10. Which regulation is least represented across all matches?

## 💬 SECTION F: Chatbot Behavior & Accuracy Testing

### Goal: Evaluate the ****chat interface****, response coherence, hallucinations, and memory use.

1. Summarize the compliance findings from the uploaded documents.
2. List everything we’re doing wrong based on current control documents.
3. Do we have enough controls for regulatory audit readiness?
4. What’s the AI’s confidence in our current data protection policies?
5. How would you prioritize fixing the gaps in the uploaded controls?
6. Give a short compliance checklist based on what we uploaded.
7. What should I tell the auditor about our missing SEBI requirements?
8. Which policy do you think is outdated based on the regulatory match?
9. Could you write a one-page compliance overview report for leadership?
10. If this were a real company, would we pass a compliance review?

## ✅ Testing Tips

To **maximize impact while testing**, follow this process:

* Upload two good, realistic files (control + regulation).
* Use 10–15 prompts per session to test **different functionalities**.
* When something fails or feels vague:  
  🔍 **Note the prompt**,  
  ❌ **Check if AI hallucinated**,  
  ✅ **Improve clause extraction or grounding logic**,  
  🛠️ **Correct LLaMA prompt or fallback to deterministic method**.

.venv\Scripts\activate

pip install -r requirements.txt

streamlit run app/dashboard.py

PS C:\Users\rishi\OneDrive\Desktop\checkmate Copy 1.1> .\.venv\Scripts\activate

(.venv) PS C:\Users\rishi\OneDrive\Desktop\checkmate Copy 1.1> python

Python 3.10.9 (tags/v3.10.9:1dd9be6, Dec 6 2022, 20:01:21) [MSC v.1934 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

>>>

>>> from api.llama\_chat\_agent import ask\_llama

>>> ask\_llama("Summarize GDPR Article 6 in simple terms.")

"GDPR Article 6 explains the legal bases for processing personal data. In simple terms, it outlines thean collect and use someone's personal information.\n\nHere are the 6 legal bases:\n\n1. \*\*Consent\*\*: Thssion for their data to be processed.\n2. \*\*Contract\*\*: Processing is necessary to fulfill a contract woviding a service they've requested.\n3. \*\*Legal Obligation\*\*: The organization is required by law to pting taxes or complying with regulatory requirements.\n4. \*\*Vital Interests\*\*: Processing is necessary hysical well-being.\n5. \*\*Public Interest\*\*: Processing is necessary for the public interest, such as scal purposes.\n6. \*\*Legitimate Interests\*\*: The organization has a legitimate reason to process the dat IT security, as long as it doesn't override the individual's rights and freedoms.\n\nIn summary, Artics for organizations to process personal data, ensuring that individuals' privacy is respected and prote

>>>

VERSION CONTROL : 11-07-2025

DOCUMENT\_PARSER.PY

# api/document\_parser.py

import os

import re

import time

from datetime import datetime

import pandas as pd

import nltk

from nltk.tokenize import sent\_tokenize, word\_tokenize

from nltk.corpus import stopwords

# NLTK downloads (only if needed)

try:

    nltk.data.find("tokenizers/punkt")

except LookupError:

    nltk.download("punkt")

try:

    nltk.data.find("corpora/stopwords")

except LookupError:

    nltk.download("stopwords")

import fitz  # PyMuPDF

from docx import Document

SUPPORTED\_EXTENSIONS = [".pdf", ".docx", ".txt", ".csv", ".xlsx", ".xls"]

stop\_words = set(stopwords.words("english"))

KNOWN\_REGULATIONS = ["GDPR", "ISO27001", "RBI", "SEBI", "PDPB", "DPDP", "MSA"]

# --- File Handlers ---

def extract\_text(file\_path):

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

    if ext not in SUPPORTED\_EXTENSIONS:

        raise ValueError(f"❌ Unsupported file format: {ext}")

    try:

        if ext == ".pdf":

            with fitz.open(file\_path) as doc:

                return "\n".join([page.get\_text() for page in doc])

        elif ext == ".docx":

            doc = Document(file\_path)

            return "\n".join([para.text for para in doc.paragraphs])

        elif ext == ".txt":

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

                return f.read()

        elif ext == ".csv":

            df = pd.read\_csv(file\_path, encoding="utf-8", on\_bad\_lines="skip")

            return "\n".join([" ".join(map(str, row)).strip() for row in df.fillna("").values.tolist()])

        elif ext in [".xlsx", ".xls"]:

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

            return "\n".join(df.astype(str).fillna("").values.flatten())

    except Exception as e:

        raise Exception(f"⚠️ Error reading {file\_path}: {str(e)}")

# --- Clause Splitter ---

def split\_into\_clauses(text, remove\_stopwords=False, doc\_name="", regulation\_name="Unknown Regulation"):

    clauses = []

    sentences = sent\_tokenize(text)

    for i, sentence in enumerate(sentences):

        s = re.sub(r"[^A-Za-z0-9\s,.()\-–/]", "", sentence.strip())

        s = re.sub(r"\s+", " ", s)

        if any(skip in s.lower() for skip in ["table of contents", "annexure", "appendix"]):

            continue

        if remove\_stopwords:

            tokens = [w for w in word\_tokenize(s) if w.lower() not in stop\_words]

            s = " ".join(tokens)

        if len(s) >= 20 and any(c.isalpha() for c in s):

            clauses.append({

                "clause\_id": f"{doc\_name}-Clause-{i+1}" if doc\_name else f"Clause-{i+1}",

                "text": s,

                "regulation": regulation\_name,

                "source\_type": "Control" if "CONTROL" in doc\_name.upper() else "Regulation"

            })

    return clauses

# --- File Saving ---

def save\_uploaded\_file(uploaded\_file, save\_dir="data/uploads"):

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

    safe\_filename = re.sub(r"[^A-Za-z0-9\_\-\.]", "\_", uploaded\_file.name)

    file\_path = os.path.join(save\_dir, safe\_filename)

    with open(file\_path, "wb") as f:

        f.write(uploaded\_file.getbuffer())

    return file\_path

def save\_text\_and\_metadata(text, original\_filename, save\_dir="data/texts"):

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

    base = os.path.splitext(original\_filename)[0]

    safe\_base = re.sub(r"[^A-Za-z0-9\_\-]", "\_", base)

    text\_path = os.path.join(save\_dir, f"{safe\_base}.txt")

    meta\_path = os.path.join(save\_dir, f"{safe\_base}\_meta.txt")

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

        f.write(text)

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

        f.write(f"📂 File: {original\_filename}\n")

        f.write(f"📦 Size (KB): {len(text.encode('utf-8')) // 1024}\n")

        f.write(f"⏱️ Processed on: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n")

    return text\_path, meta\_path

# --- Full Pipeline ---

def process\_uploaded\_file(uploaded\_file, save\_dir="data/uploads", remove\_stopwords=False):

    start = time.time()

    file\_path = save\_uploaded\_file(uploaded\_file, save\_dir)

    print(f"[TIME] File saved: {time.time() - start:.2f}s")

    raw\_text = extract\_text(file\_path)

    print(f"[TIME] Text extracted: {time.time() - start:.2f}s")

    save\_text\_and\_metadata(raw\_text, uploaded\_file.name, save\_dir="data/texts")

    print(f"[TIME] Metadata saved: {time.time() - start:.2f}s")

    base\_name = os.path.splitext(uploaded\_file.name)[0].upper()

    matched\_reg = next((r for r in KNOWN\_REGULATIONS if r in base\_name), "Uploaded Document")

    clauses = split\_into\_clauses(

        raw\_text,

        remove\_stopwords=remove\_stopwords,

        doc\_name=base\_name,

        regulation\_name=matched\_reg

    )

    print(f"[TIME] Clauses split: {time.time() - start:.2f}s")

    print(f"[TOTAL TIME] Processed {uploaded\_file.name} in {time.time() - start:.2f}s\n")

    return clauses

LLAMA\_CHAT\_AGENT.PY

# api/llama\_chat\_agent.py

import re

import os

import streamlit as st

from typing import List, Dict, Generator, Union

from dotenv import load\_dotenv

from openai import OpenAI

# Load API key

load\_dotenv()

api\_key = os.getenv("GROQ\_API\_KEY")

if not api\_key:

    raise ValueError("🚨 GROQ\_API\_KEY not found. Check your .env file.")

client = OpenAI(

    base\_url="https://api.groq.com/openai/v1",

    api\_key=api\_key

)

def approximate\_token\_count(text: str) -> int:

    tokens = re.findall(r"\w+|[^\w\s]", text, re.UNICODE)

    return len(tokens)

def dynamic\_max\_tokens(messages: List[Dict], model\_max\_tokens: int = 8192) -> int:

    used = sum(approximate\_token\_count(m.get("content", "")) for m in messages)

    return max(model\_max\_tokens - used - 100, 100)

def get\_memory\_context() -> Union[Dict, None]:

    """

    Builds a context summary from matched/missing clauses for injection into LLaMA.

    """

    if not st.session\_state.get("processed\_data"):

        return None

    matched = st.session\_state["processed\_data"].get("matched", [])[:5]

    missing = st.session\_state["processed\_data"].get("missing", [])[:3]

    context\_lines = []

    if matched:

        context\_lines.append("✅ Top Matched Clauses:")

        for item in matched:

            clause\_id = item.get("Clause ID", "—")

            match\_type = item.get("Match Type", "—")

            reg = item.get("Regulation", "—")

            text = item.get("Control Clause", "")[:70].strip()

            context\_lines.append(f"- [{clause\_id}] ({match\_type}) in `{reg}`: \"{text}...\"")

    if missing:

        context\_lines.append("❌ Top Missing Clauses:")

        for item in missing:

            text = item.get("Missing Clause", "")[:70].strip()

            gap = item.get("gap", "")

            context\_lines.append(f"- \"{text}...\"  (Gap: {gap})")

    full\_context = "\n".join(context\_lines)

    return {

        "role": "system",

        "content": (

            "You are a legal compliance expert AI assistant. Explain compliance results from matched and missing clauses. "

            "Use Clause IDs, Match Types, and Regulations where relevant. Provide clear, risk-based reasoning."

            "\n\n📄 Match Summary:\n" + full\_context

        )

    }

def inject\_context\_once(chat\_history: List[Dict]) -> List[Dict]:

    """

    Injects match/missing clause memory only once per session.

    Doesn't appear in UI; just for LLaMA grounding.

    """

    if not st.session\_state.get("context\_injected", False):

        context = get\_memory\_context()

        if context:

            st.session\_state.context\_injected = True

            return [context] + [m for m in chat\_history if m["role"] != "system"]

    return chat\_history

def ask\_llama(

    messages: Union[str, List[Dict]],

    model: str = "llama3-70b-8192",

    temperature: float = 0.2,

    stream: bool = False,

    system\_instructions: str = None

) -> Union[str, Generator[str, None, None]]:

    if isinstance(messages, str):

        messages = [{"role": "user", "content": messages}]

    # Inject context summary if not yet injected

    messages = inject\_context\_once(messages)

    max\_tokens = dynamic\_max\_tokens(messages)

    params = {

        "model": model,

        "messages": messages,

        "max\_tokens": max\_tokens,

        "temperature": temperature,

        "stream": stream

    }

    try:

        if stream:

            response\_stream = client.chat.completions.create(\*\*params)

            return (chunk.choices[0].delta.content or "" for chunk in response\_stream)

        else:

            response = client.chat.completions.create(\*\*params)

            content = response.choices[0].message.content.strip()

            if response.choices[0].finish\_reason != "stop":

                content += "\n\n⚠️ Response may be incomplete. Consider increasing `max\_tokens`."

            return content

    except Exception as e:

        return f"[LLaMA error: {str(e)}]"

def get\_flashcard\_prompts\_from\_context() -> List[str]:

    """

    Generate 5 flashcard-style prompts based on uploaded files + match/missing clause context.

    These prompts are designed to guide the user into asking deeper questions.

    """

    prompts = []

    if not st.session\_state.get("processed\_data"):

        return prompts

    matched = st.session\_state["processed\_data"].get("matched", [])[:3]

    missing = st.session\_state["processed\_data"].get("missing", [])[:2]

    if matched:

        for item in matched:

            cid = item.get("Clause ID", "Clause")

            reg = item.get("Regulation", "a regulation")

            prompts.append(f"💡 How does \*\*{cid}\*\* align with {reg}?")

            prompts.append(f"🔎 Can we improve compliance for \*\*{cid}\*\* to reduce risk?")

            break  # Only one matched clause for generalization

    if missing:

        for item in missing:

            gap = item.get("gap", "—")

            text = item.get("Missing Clause", "")[:50]

            prompts.append(f"⚠️ Why might the following clause be flagged as missing? “{text}...”")

            prompts.append(f"📘 Suggest a control clause to fill the gap: {gap}")

            break  # Only one example to keep general

    prompts.append("🧠 What’s the overall compliance score and which areas need the most attention?")

    return prompts

MATCH\_ENGINE.PY

# api/match\_engine.py

import os

import re

import numpy as np

from nltk.corpus import stopwords

from sentence\_transformers import SentenceTransformer, util

from concurrent.futures import ThreadPoolExecutor

import streamlit as st

# --- Model & Config ---

model = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")

THRESH\_STRONG = 0.75

THRESH\_PARTIAL = 0.5

THRESH\_WEAK = 0.25

STOPWORDS = set(stopwords.words("english"))

# --- Optional LLaMA Analysis ---

USE\_LLaMA = True

MAX\_LLaMA\_ANALYSIS = st.session\_state.get("max\_llama\_clauses", 3)  # 👈 Control via UI slider

GROQ\_API\_KEY = os.getenv("GROQ\_API\_KEY")

GROQ\_URL = "https://api.groq.com/openai/v1/chat/completions"

LLAMA\_MODEL = "llama3-70b-8192"

@st.cache\_data(show\_spinner=False)

def batch\_encode(texts):

    return model.encode(texts, convert\_to\_tensor=True, show\_progress\_bar=False)

def clean\_text(text):

    return re.sub(r'\s+', ' ', text.strip())

def extract\_words(text, remove\_stopwords=True):

    words = set(re.findall(r'\w+', text.lower()))

    return words - STOPWORDS if remove\_stopwords else words

def classify\_status(score):

    if score >= THRESH\_STRONG:

        return "Strong Match"

    elif score >= THRESH\_PARTIAL:

        return "Partial Match"

    elif score >= THRESH\_WEAK:

        return "Weak Match"

    else:

        return "Unmatched"

def generate\_llama\_analysis(control, regulation, score, reg\_name):

    import httpx

    prompt = f"""

Control Clause:

\"\"\"{control}\"\"\"

Regulation Clause from {reg\_name}:

\"\"\"{regulation}\"\"\"

Match score: {round(score, 3)}

You are a compliance analyst. Analyze the above in detail:

1. Classify the match: Strong, Partial, Weak, Unmatched

2. 🔗 Overlap

3. 📉 Gaps

4. 🧠 Rewrite (if needed)

5. ⚠️ Non-Compliance Risk

6. 💸 Estimated Fine (Low, Medium, High + reason)

"""

    headers = {

        "Authorization": f"Bearer {GROQ\_API\_KEY}",

        "Content-Type": "application/json"

    }

    payload = {

        "model": LLAMA\_MODEL,

        "temperature": 0.4,

        "max\_tokens": 300,

        "messages": [{"role": "user", "content": prompt}]

    }

    try:

        response = httpx.post(GROQ\_URL, json=payload, headers=headers, timeout=45.0)

        response.raise\_for\_status()

        return response.json()["choices"][0]["message"]["content"].strip()

    except Exception as e:

        return f"[LLaMA Error: {str(e)}]"

def match\_single\_clause(i, clause, reg\_texts, reg\_embeddings, remove\_stopwords=True):

    clause\_text = clause.get("text", "").strip() if isinstance(clause, dict) else str(clause).strip()

    clause\_id = clause.get("clause\_id", f"Control-{i+1}") if isinstance(clause, dict) else f"Control-{i+1}"

    if not clause\_text:

        return [{

            "control\_id": clause\_id,

            "control": "[EMPTY]",

            "status": "Unmatched",

            "matched\_clause": "—",

            "regulation": "—",

            "score": 0.0,

            "overlap": "—",

            "gap": "Clause was empty",

            "reason": "—",

            "rewrite": "—",

            "risk": "—",

            "fine": "—"

        }]

    cleaned\_control = clean\_text(clause\_text)

    control\_embedding = model.encode(cleaned\_control, convert\_to\_tensor=True)

    similarities = util.cos\_sim(control\_embedding, reg\_embeddings)[0].cpu().numpy()

    best\_idx = similarities.argmax()

    best\_score = float(similarities[best\_idx])

    reg\_clause = reg\_texts[best\_idx]

    reg\_text\_clean = clean\_text(reg\_clause["text"])

    overlap = sorted(extract\_words(cleaned\_control) & extract\_words(reg\_text\_clean))

    missing = sorted(extract\_words(reg\_text\_clean) - extract\_words(cleaned\_control))

    should\_use\_llama = USE\_LLaMA and i < MAX\_LLaMA\_ANALYSIS

    reason = generate\_llama\_analysis(cleaned\_control, reg\_clause["text"], best\_score, reg\_clause["regulation"]) if should\_use\_llama else "AI analysis not applied."

    return [{

        "control\_id": clause\_id,

        "control": cleaned\_control,

        "status": classify\_status(best\_score),

        "matched\_clause": reg\_clause["text"],

        "regulation": reg\_clause["regulation"],

        "score": round(best\_score, 3),

        "overlap": ", ".join(overlap) if overlap else "No overlap",

        "gap": ", ".join(missing) if missing else "None",

        "reason": reason,

        "rewrite": "See AI reasoning" if should\_use\_llama else "—",

        "risk": "See AI reasoning" if should\_use\_llama else "—",

        "fine": "See AI reasoning" if should\_use\_llama else "—"

    }]

def process\_and\_match\_multiple\_docs(control\_clauses, regulation\_clauses, remove\_stopwords=True):

    reg\_texts = [{

        "text": clean\_text(rc.get("text", "")),

        "regulation": rc.get("regulation", "Unknown Regulation")

    } for rc in regulation\_clauses]

    reg\_embeddings = batch\_encode([r["text"] for r in reg\_texts])

    results = []

    with ThreadPoolExecutor() as executor:

        futures = [

            executor.submit(match\_single\_clause, i, clause, reg\_texts, reg\_embeddings, remove\_stopwords)

            for i, clause in enumerate(control\_clauses)

        ]

        for future in futures:

            results.extend(future.result())

    return results

REPORT\_BUILDER.PY

# api/report\_builder.py

import io

import pandas as pd

from datetime import datetime

def clean\_field(val):

    """Clean hallucinated or broken AI responses."""

    if isinstance(val, str) and any(err in val for err in ["LLaMA Error", "Too Many Requests", "Payload Too Large"]):

        return "⚠️ Unable to generate reasoning (AI overload)"

    return val if val else "—"

def generate\_final\_csv\_report(matched\_controls, missing\_controls, chat\_history=None):

    """

    Generates a two-sheet Excel file:

    1. Control-Gap Analysis

    2. Chatbot History

    """

    output = io.BytesIO()

    # --- Helper: Clean and Structure Clause Data ---

    def preprocess(df\_raw, is\_missing=False):

        df = pd.DataFrame(df\_raw).copy()

        # Handle missing columns safely

        df["Clause ID"] = df["Clause ID"] if "Clause ID" in df.columns else "—"

        df["Control Clause"] = df["Control Clause"] if "Control Clause" in df.columns else df.get("Missing Clause", "—")

        df["Match Type"] = "Unmatched" if is\_missing else df.get("Match Type", "—")

        df["Regulation"] = df.get("Regulation", "—")

        df["Match Score (%)"] = df.get("Score", 0.0)

        df["Overlap Terms"] = df.get("Overlap Terms", "—")

        df["Semantic Gap Analysis"] = df.get("Gap", df.get("gap", "—"))

        # Clean AI Reasoning

        if "Reasoning" in df.columns:

            df["AI Reasoning"] = df["Reasoning"].apply(clean\_field)

        elif "reason" in df.columns:

            df["AI Reasoning"] = df["reason"].apply(clean\_field)

        else:

            df["AI Reasoning"] = "—"

        df["Suggested Rewrite for Better Compliance"] = df.get("rewrite", "—")

        df["Associated Risks"] = df.get("risk", "—")

        df["Potential Fines"] = df.get("fine", "—")

        # Final column structure

        return df[[

            "Clause ID",

            "Control Clause",

            "Match Type",

            "Regulation",

            "Match Score (%)",

            "Overlap Terms",

            "Semantic Gap Analysis",

            "AI Reasoning",

            "Suggested Rewrite for Better Compliance",

            "Associated Risks",

            "Potential Fines"

        ]]

    # Process both matched and unmatched

    df\_matched = preprocess(matched\_controls)

    df\_missing = preprocess(missing\_controls, is\_missing=True)

    df\_combined = pd.concat([df\_matched, df\_missing], ignore\_index=True)

    # --- Chatbot History Sheet ---

    chat\_rows = []

    if chat\_history:

        for msg in chat\_history:

            chat\_rows.append({

                "Role": msg.get("role", "Unknown").title(),

                "Message": msg.get("content", "").replace("\n", " ").strip(),

                "Timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S")

            })

    df\_chat = pd.DataFrame(chat\_rows or [], columns=["Role", "Message", "Timestamp"])

    # --- Write to Excel ---

    with pd.ExcelWriter(output, engine="xlsxwriter") as writer:

        df\_combined.to\_excel(writer, sheet\_name="Control-Gap Analysis", index=False)

        df\_chat.to\_excel(writer, sheet\_name="Chatbot History", index=False)

    output.seek(0)

    return output

DASHBOARD.PY

# app/dashboard.py

import os

import sys

import logging

from pathlib import Path

import streamlit as st

# Add project root to sys.path

sys.path.append(str(Path(\_\_file\_\_).parent.parent.resolve()))

# Import app modules

from api.document\_parser import process\_uploaded\_file

from api.match\_engine import process\_and\_match\_multiple\_docs

from api.report\_builder import generate\_final\_csv\_report

from api.llama\_chat\_agent import ask\_llama, get\_flashcard\_prompts\_from\_context

# Logging setup

logging.basicConfig(level=logging.INFO)

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

# App config

st.set\_page\_config(page\_title="Compliance Intelligence Engine", layout="wide")

# 🎨 Dark Mode Styling

st.markdown("""

<style>

html, body, [data-testid="stAppViewContainer"] {

    background-color: #0e1117;

    color: #ffffff;

}

[data-testid="stSidebar"] {

    background-color: #1e1e1e;

}

.stTextInput > div > input {

    color: white; background-color: #262730;

}

.stButton > button {

    color: white; background-color: #2a2d3a;

}

thead tr th, tbody tr td {

    color: white !important;

}

</style>

""", unsafe\_allow\_html=True)

# Session state

if "chat\_history" not in st.session\_state:

    st.session\_state.chat\_history = []

if "processed\_data" not in st.session\_state:

    st.session\_state.processed\_data = None

if "context\_injected" not in st.session\_state:

    st.session\_state.context\_injected = False

# 📂 File Upload Section

st.sidebar.header("📂 Upload Documents")

control\_docs = st.sidebar.file\_uploader("Upload Company Control Repository", type=["pdf", "docx", "txt", "csv", "xlsx"], accept\_multiple\_files=True)

regulation\_docs = st.sidebar.file\_uploader("Upload Regulatory Documents (Multiple)", type=["pdf", "docx", "txt", "csv", "xlsx"], accept\_multiple\_files=True)

if not control\_docs or not regulation\_docs:

    st.sidebar.warning("Please upload both control and regulatory documents.")

    st.stop()

# 🧠 AI Matching Engine

def run\_matching():

    try:

        control\_clauses = []

        regulation\_clauses = []

        for f in control\_docs:

            control\_clauses += process\_uploaded\_file(f)

        for f in regulation\_docs:

            clauses = process\_uploaded\_file(f)

            regulation\_clauses += [{"text": c["text"], "regulation": f.name} for c in clauses]

        results = process\_and\_match\_multiple\_docs(control\_clauses, regulation\_clauses)

        matched, missing = [], []

        for r in results:

            score = r.get("score", 0.0)

            if r["status"] == "Unmatched":

                missing.append({

                    "Missing Clause": r.get("control", "—"),

                    "Score": score,

                    "gap": r.get("gap", "—"),

                    "reason": r.get("reason", "—"),

                    "Regulation": r.get("regulation", "—"),

                    "rewrite": r.get("rewrite", "—"),

                    "risk": r.get("risk", "—"),

                    "fine": r.get("fine", "—")

                })

            else:

                matched.append({

                    "Clause ID": r.get("control\_id", "—"),

                    "Control Clause": r.get("control", "—"),

                    "Match Type": r.get("status", "—"),

                    "Score": score,

                    "Regulation": r.get("regulation", "—"),

                    "Overlap Terms": r.get("overlap", "—"),

                    "Gap": r.get("gap", "—"),

                    "Reasoning": r.get("reason", "—"),

                    "rewrite": r.get("rewrite", "—"),

                    "risk": r.get("risk", "—"),

                    "fine": r.get("fine", "—")

                })

        return {"matched": matched, "missing": missing}

    except Exception as e:

        st.error("❌ AI Matching failed.")

        logger.exception(e)

        return {"matched": [], "missing": []}

# ▶️ Run Matching Button

if st.sidebar.button("🔍 Run Compliance Matching"):

    with st.spinner("Running AI-based clause analysis..."):

        st.session\_state.processed\_data = run\_matching()

# 📥 Export CSV Report

if st.sidebar.button("📥 Download CSV Report"):

    try:

        report = generate\_final\_csv\_report(

            st.session\_state.get("processed\_data", {}).get("matched", []),

            st.session\_state.get("processed\_data", {}).get("missing", []),

            chat\_history=st.session\_state.get("chat\_history", [])

        )

        st.sidebar.download\_button(

            label="📊 Export CSV Report",

            data=report,

            file\_name="Compliance\_Report.xlsx",

            mime="application/vnd.openxmlformats-officedocument.spreadsheetml.sheet"

        )

    except Exception as e:

        st.sidebar.error("❌ Report generation failed.")

        logger.error(e)

# 💬 Chatbot Interface

st.markdown("## 💬 Chat with Compliance Assistant")

# 📊 Show Matching Summary

if st.session\_state.get("processed\_data"):

    st.markdown("### 📄 Summary of Matching Results")

    top\_matches = st.session\_state["processed\_data"].get("matched", [])[:3]

    top\_misses = st.session\_state["processed\_data"].get("missing", [])[:2]

    for m in top\_matches:

        st.info(f"✅ `{m['Clause ID']}` in \*\*{m['Regulation']}\*\* matched with score \*\*{round(m['Score'], 2)}\*\*")

    for m in top\_misses:

        st.warning(f"⚠️ Missing Clause: \*{m['Missing Clause'][:60]}...\* — Possible Gap: \*\*{m['gap']}\*\*")

# 📎 Suggested Flashcard Prompts

st.markdown("### 🧠 Suggested Flashcard Prompts")

prompts = get\_flashcard\_prompts\_from\_context()

if not prompts:

    st.info("Upload documents and run matching to generate prompts.")

else:

    for prompt in prompts:

        if st.button(prompt, key=f"prompt\_{prompt}"):

            st.session\_state.chat\_history.append({"role": "user", "content": prompt})

            try:

                with st.spinner("Thinking..."):

                    response\_stream = ask\_llama(st.session\_state.chat\_history, stream=True)

                    full\_response = ""

                    for chunk in response\_stream:

                        full\_response += chunk

                        st.write\_stream(iter([chunk]))

                st.session\_state.chat\_history.append({"role": "assistant", "content": full\_response})

            except Exception as e:

                st.error("❌ Chat failed.")

                logger.error(e)

# 📝 User Chat Form

with st.form("chat\_form", clear\_on\_submit=True):

    user\_input = st.text\_input("Ask anything about compliance, gaps, controls or risks:")

    submitted = st.form\_submit\_button("📨 Send")

if submitted and user\_input:

    st.session\_state.chat\_history.append({"role": "user", "content": user\_input})

    try:

        with st.spinner("Thinking..."):

            response\_stream = ask\_llama(st.session\_state.chat\_history, stream=True)

            full\_response = ""

            for chunk in response\_stream:

                full\_response += chunk

                st.write\_stream(iter([chunk]))

        st.session\_state.chat\_history.append({"role": "assistant", "content": full\_response})

    except Exception as e:

        st.error("❌ Chat failed.")

        logger.error(e)

# 💬 Chat History with Popups

for msg in st.session\_state.chat\_history:

    with st.chat\_message(msg["role"]):

        st.markdown(msg["content"])

        if "matched" in st.session\_state.get("processed\_data", {}):

            for m in st.session\_state["processed\_data"]["matched"]:

                if m["Clause ID"] in msg["content"]:

                    with st.expander(f"📄 Show Clause: {m['Clause ID']}"):

                        st.code(m["Control Clause"])

                        st.markdown(f"🔗 [Reference Regulation: {m['Regulation']}]({f'#'})")

# ♻️ Reset All

if st.sidebar.button("🧹 Reset Chat & Results"):

    st.session\_state.chat\_history = []

    st.session\_state.context\_injected = False

    st.session\_state.processed\_data = None

    st.rerun()

VERSION CONTROL 2 : IN 1.2

DOCUMENT PARSER

# api/document\_parser.py

import os

import re

import time

from datetime import datetime

import pandas as pd

import nltk

from nltk.tokenize import sent\_tokenize, word\_tokenize

from nltk.corpus import stopwords

# NLTK checks

try:

    nltk.data.find("tokenizers/punkt")

except LookupError:

    nltk.download("punkt")

try:

    nltk.data.find("corpora/stopwords")

except LookupError:

    nltk.download("stopwords")

import fitz  # PyMuPDF

from docx import Document

SUPPORTED\_EXTENSIONS = [".pdf", ".docx", ".txt", ".csv", ".xlsx", ".xls"]

stop\_words = set(stopwords.words("english"))

KNOWN\_REGULATIONS = ["GDPR", "ISO27001", "RBI", "SEBI", "PDPB", "DPDP", "MSA"]

# --- File Handlers ---

def extract\_text(file\_path):

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

    if ext not in SUPPORTED\_EXTENSIONS:

        raise ValueError(f"❌ Unsupported file format: {ext}")

    try:

        if ext == ".pdf":

            return extract\_pdf\_clauses(file\_path)

        elif ext == ".docx":

            return extract\_docx\_clauses(file\_path)

        elif ext == ".txt":

            return extract\_txt\_clauses(file\_path)

        elif ext == ".csv":

            return extract\_csv\_clauses(file\_path)

        elif ext in [".xlsx", ".xls"]:

            return extract\_excel\_clauses(file\_path)

    except Exception as e:

        raise Exception(f"⚠️ Error reading {file\_path}: {str(e)}")

# --- Clause Extractors by Format ---

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

    clauses = []

    with fitz.open(file\_path) as doc:

        for page\_num, page in enumerate(doc, start=1):

            text = page.get\_text()

            sections = sent\_tokenize(text)

            for i, sent in enumerate(sections):

                sent = sanitize\_clause(sent)

                if is\_valid\_clause(sent):

                    clauses.append({

                        "clause\_id": f"{os.path.basename(file\_path)}-P{page\_num}-C{i+1}",

                        "text": sent,

                        "doc\_name": os.path.basename(file\_path),

                        "page\_num": page\_num,

                        "section": f"Page {page\_num}",

                        "source\_type": infer\_source\_type(file\_path)

                    })

    return clauses

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

    clauses = []

    doc = Document(file\_path)

    current\_section = ""

    para\_count = 0

    for para in doc.paragraphs:

        text = para.text.strip()

        if para.style.name.startswith("Heading"):

            current\_section = text

            continue

        if len(text) < 20:

            continue

        para\_count += 1

        sent = sanitize\_clause(text)

        if is\_valid\_clause(sent):

            clauses.append({

                "clause\_id": f"{os.path.basename(file\_path)}-S{para\_count}",

                "text": sent,

                "doc\_name": os.path.basename(file\_path),

                "page\_num": None,

                "section": current\_section or "Untitled Section",

                "source\_type": infer\_source\_type(file\_path)

            })

    return clauses

def extract\_txt\_clauses(file\_path):

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

        text = f.read()

    return split\_sentences\_into\_clauses(text, file\_path)

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

    df = pd.read\_csv(file\_path, encoding="utf-8", on\_bad\_lines="skip")

    clauses = []

    for i, row in df.iterrows():

        joined = " ".join(map(str, row.dropna())).strip()

        joined = sanitize\_clause(joined)

        if is\_valid\_clause(joined):

            clauses.append({

                "clause\_id": f"{os.path.basename(file\_path)}-R{i+1}",

                "text": joined,

                "doc\_name": os.path.basename(file\_path),

                "page\_num": None,

                "section": "Row Data",

                "source\_type": infer\_source\_type(file\_path)

            })

    return clauses

def extract\_excel\_clauses(file\_path):

    df = pd.read\_excel(file\_path)

    clauses = []

    for i, row in df.iterrows():

        joined = " ".join(map(str, row.dropna())).strip()

        joined = sanitize\_clause(joined)

        if is\_valid\_clause(joined):

            clauses.append({

                "clause\_id": f"{os.path.basename(file\_path)}-XL{i+1}",

                "text": joined,

                "doc\_name": os.path.basename(file\_path),

                "page\_num": None,

                "section": "Sheet Data",

                "source\_type": infer\_source\_type(file\_path)

            })

    return clauses

def split\_sentences\_into\_clauses(text, file\_path):

    clauses = []

    for i, sent in enumerate(sent\_tokenize(text)):

        sent = sanitize\_clause(sent)

        if is\_valid\_clause(sent):

            clauses.append({

                "clause\_id": f"{os.path.basename(file\_path)}-T{i+1}",

                "text": sent,

                "doc\_name": os.path.basename(file\_path),

                "page\_num": None,

                "section": "Text File",

                "source\_type": infer\_source\_type(file\_path)

            })

    return clauses

# --- Utilities ---

def sanitize\_clause(clause):

    s = re.sub(r"[^A-Za-z0-9\s,.()\-–/]", "", clause.strip())

    s = re.sub(r"\s+", " ", s)

    return s

def is\_valid\_clause(text):

    if any(skip in text.lower() for skip in ["table of contents", "annexure", "appendix"]):

        return False

    return len(text) >= 20 and any(c.isalpha() for c in text)

def infer\_source\_type(file\_path):

    fname = os.path.basename(file\_path).upper()

    return "Control" if "CONTROL" in fname else "Regulation"

# --- Save Raw File/Text ---

def save\_uploaded\_file(uploaded\_file, save\_dir="data/uploads"):

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

    safe\_filename = re.sub(r"[^A-Za-z0-9\_\-\.]", "\_", uploaded\_file.name)

    file\_path = os.path.join(save\_dir, safe\_filename)

    with open(file\_path, "wb") as f:

        f.write(uploaded\_file.getbuffer())

    return file\_path

def save\_text\_and\_metadata(text, original\_filename, save\_dir="data/texts"):

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

    base = os.path.splitext(original\_filename)[0]

    safe\_base = re.sub(r"[^A-Za-z0-9\_\-]", "\_", base)

    text\_path = os.path.join(save\_dir, f"{safe\_base}.txt")

    meta\_path = os.path.join(save\_dir, f"{safe\_base}\_meta.txt")

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

        f.write(text)

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

        f.write(f"📂 File: {original\_filename}\n")

        f.write(f"📦 Size (KB): {len(text.encode('utf-8')) // 1024}\n")

        f.write(f"⏱️ Processed on: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n")

    return text\_path, meta\_path

# --- Main Pipeline ---

def process\_uploaded\_file(uploaded\_file, save\_dir="data/uploads"):

    start = time.time()

    file\_path = save\_uploaded\_file(uploaded\_file, save\_dir)

    print(f"[TIME] File saved: {time.time() - start:.2f}s")

    clauses = extract\_text(file\_path)

    print(f"[TIME] Text extracted and split: {time.time() - start:.2f}s")

    flat\_text = "\n".join([c["text"] for c in clauses])

    save\_text\_and\_metadata(flat\_text, uploaded\_file.name, save\_dir="data/texts")

    print(f"[TIME] Metadata saved: {time.time() - start:.2f}s")

    print(f"[TOTAL TIME] Processed {uploaded\_file.name} in {time.time() - start:.2f}s\n")

    return clauses

LLAMA CHAT AGENT

# api/llama\_chat\_agent.py

import re

import os

import streamlit as st

from typing import List, Dict, Generator, Union

from dotenv import load\_dotenv

from openai import OpenAI

# Load API key from .env

load\_dotenv()

api\_key = os.getenv("GROQ\_API\_KEY")

if not api\_key:

    raise ValueError("🚨 GROQ\_API\_KEY not found. Check your .env file.")

client = OpenAI(

    base\_url="https://api.groq.com/openai/v1",

    api\_key=api\_key

)

# --- Utility: Token Counting ---

def approximate\_token\_count(text: str) -> int:

    return len(re.findall(r"\w+|[^\w\s]", text, re.UNICODE))

def dynamic\_max\_tokens(messages: List[Dict], model\_max\_tokens: int = 8192) -> int:

    used = sum(approximate\_token\_count(m.get("content", "")) for m in messages)

    return max(model\_max\_tokens - used - 150, 150)

# --- Match Summary Context Injection ---

def get\_memory\_context() -> Union[Dict, None]:

    if not st.session\_state.get("processed\_data"):

        return None

    matched = st.session\_state["processed\_data"].get("matched", [])[:5]

    missing = st.session\_state["processed\_data"].get("missing", [])[:3]

    context\_lines = []

    if matched:

        context\_lines.append("✅ Top Matched Clauses:")

        for item in matched:

            cid = item.get("control\_id", "Clause")

            reg = item.get("regulation", "Regulation")

            score = item.get("score", "—")

            clause\_text = item.get("control", "")[:80]

            context\_lines.append(f"- `{cid}` in \*\*{reg}\*\* (Score: {score}): \"{clause\_text}...\"")

    if missing:

        context\_lines.append("❌ Missing Clauses:")

        for item in missing:

            clause\_text = item.get("control", "")[:70]

            gap = item.get("gap", "—")

            context\_lines.append(f"- \"{clause\_text}...\"  (Gap: {gap})")

    return {

        "role": "system",

        "content": (

            "You are an AI legal compliance assistant. Explain gaps, overlaps, and risks based on control and regulation clauses."

            "\n\n📄 Clause Summary:\n" + "\n".join(context\_lines)

        )

    }

def inject\_context\_once(chat\_history: List[Dict]) -> List[Dict]:

    if not st.session\_state.get("context\_injected", False):

        context = get\_memory\_context()

        if context:

            st.session\_state.context\_injected = True

            return [context] + [m for m in chat\_history if m["role"] != "system"]

    return chat\_history

# --- Chat Function ---

def ask\_llama(

    messages: Union[str, List[Dict]],

    model: str = "llama3-70b-8192",

    temperature: float = 0.2,

    stream: bool = False

) -> Union[str, Generator[str, None, None]]:

    if isinstance(messages, str):

        messages = [{"role": "user", "content": messages}]

    messages = inject\_context\_once(messages)

    max\_tokens = dynamic\_max\_tokens(messages)

    params = {

        "model": model,

        "messages": messages,

        "max\_tokens": max\_tokens,

        "temperature": temperature,

        "stream": stream

    }

    try:

        if stream:

            response = client.chat.completions.create(\*\*params)

            return (chunk.choices[0].delta.content or "" for chunk in response)

        else:

            response = client.chat.completions.create(\*\*params)

            reply = response.choices[0].message.content.strip()

            if response.choices[0].finish\_reason != "stop":

                reply += "\n\n⚠️ Truncated due to token limit."

            return reply

    except Exception as e:

        return f"[LLaMA error: {str(e)}]"

# --- Flashcard Generator ---

def get\_flashcard\_prompts\_from\_context() -> List[str]:

    prompts = []

    data = st.session\_state.get("processed\_data")

    if not data:

        return prompts

    matched = data.get("matched", [])

    missing = data.get("missing", [])

    # Use only 1–2 examples to avoid clutter

    if matched:

        top = matched[0]

        cid = top.get("control\_id", "Control")

        reg = top.get("regulation", "Regulation")

        prompts.append(f"🔍 How does \*\*{cid}\*\* align with {reg}?")

        prompts.append(f"💡 Can \*\*{cid}\*\* be improved to meet full compliance?")

    if missing:

        top = missing[0]

        gap = top.get("gap", "—")

        text = top.get("control", "")[:50]

        prompts.append(f"⚠️ Why is this clause marked missing? “{text}...”")

        prompts.append(f"📘 Suggest a new control to cover: {gap}")

    prompts.append("🧠 What's the overall risk profile based on current matches?")

    return prompts

MATCH ENGINE

# api/match\_engine.py

import os

import re

import numpy as np

from nltk.corpus import stopwords

from sentence\_transformers import SentenceTransformer, util

from concurrent.futures import ThreadPoolExecutor

import streamlit as st

import httpx

import json

# --- Config ---

model = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")

THRESH\_STRONG = 0.75

THRESH\_PARTIAL = 0.5

THRESH\_WEAK = 0.25

TOP\_K = 1  # How many top matches per control clause

STOPWORDS = set(stopwords.words("english"))

USE\_LLaMA = True

MAX\_LLaMA\_ANALYSIS = st.session\_state.get("max\_llama\_clauses", 3)

GROQ\_API\_KEY = os.getenv("GROQ\_API\_KEY")

LLAMA\_MODEL = "llama3-70b-8192"

GROQ\_URL = "https://api.groq.com/openai/v1/chat/completions"

@st.cache\_data(show\_spinner=False)

def batch\_encode(texts):

    return model.encode(texts, convert\_to\_tensor=True, show\_progress\_bar=False)

def clean\_text(text):

    return re.sub(r'\s+', ' ', text.strip())

def extract\_words(text, remove\_stopwords=True):

    words = set(re.findall(r'\w+', text.lower()))

    return words - STOPWORDS if remove\_stopwords else words

def classify\_status(score):

    if score >= THRESH\_STRONG:

        return "Strong Match"

    elif score >= THRESH\_PARTIAL:

        return "Partial Match"

    elif score >= THRESH\_WEAK:

        return "Weak Match"

    else:

        return "Unmatched"

def parse\_llama\_response(response\_text):

    result = {

        "rewrite": "—",

        "overlap": "—",

        "gap": "—",

        "reason": "—",

        "risk": "—",

        "fine": "—"

    }

    try:

        # Simple line-based breakdown

        lines = response\_text.split("\n")

        for line in lines:

            l = line.lower()

            if "overlap" in l:

                result["overlap"] = line.split(":")[-1].strip()

            elif "gap" in l or "missing" in l:

                result["gap"] = line.split(":")[-1].strip()

            elif "rewrite" in l:

                result["rewrite"] = line.split(":", 1)[-1].strip()

            elif "risk" in l:

                result["risk"] = line.split(":", 1)[-1].strip()

            elif "fine" in l:

                result["fine"] = line.split(":", 1)[-1].strip()

            elif "classify" in l or "match" in l:

                result["reason"] = line.strip()

    except Exception:

        result["reason"] = response\_text.strip()

    return result

def generate\_llama\_analysis(control, regulation, score, reg\_name):

    prompt = f"""

Control Clause:

\"\"\"{control}\"\"\"

Regulation Clause from {reg\_name}:

\"\"\"{regulation}\"\"\"

Match score: {round(score, 3)}

You are a compliance analyst. Analyze the above in detail:

1. Classify the match: Strong, Partial, Weak, Unmatched

2. 🔗 Overlap

3. 📉 Gaps

4. 🧠 Rewrite (if needed)

5. ⚠️ Non-Compliance Risk

6. 💸 Estimated Fine (Low, Medium, High + reason)

Provide your answer in plain text format.

"""

    headers = {

        "Authorization": f"Bearer {GROQ\_API\_KEY}",

        "Content-Type": "application/json"

    }

    payload = {

        "model": LLAMA\_MODEL,

        "temperature": 0.4,

        "max\_tokens": 400,

        "messages": [{"role": "user", "content": prompt}]

    }

    try:

        response = httpx.post(GROQ\_URL, json=payload, headers=headers, timeout=45.0)

        response.raise\_for\_status()

        return response.json()["choices"][0]["message"]["content"].strip()

    except Exception as e:

        return f"[LLaMA Error: {str(e)}]"

def match\_single\_clause(i, control\_clause, reg\_clauses, reg\_embeddings):

    control\_text = control\_clause.get("text", "").strip()

    if not control\_text:

        return [{

            "control\_id": control\_clause.get("clause\_id", f"Control-{i+1}"),

            "control": "[EMPTY]",

            "status": "Unmatched",

            "score": 0.0,

            "matched\_clause": "—",

            "regulation": "—",

            "doc\_name": "—",

            "page\_num": "—",

            "section": "—",

            "overlap": "—",

            "gap": "Clause was empty",

            "reason": "—",

            "rewrite": "—",

            "risk": "—",

            "fine": "—"

        }]

    cleaned\_control = clean\_text(control\_text)

    control\_embedding = model.encode(cleaned\_control, convert\_to\_tensor=True)

    similarities = util.cos\_sim(control\_embedding, reg\_embeddings)[0].cpu().numpy()

    top\_indices = similarities.argsort()[-TOP\_K:][::-1]

    results = []

    for j, best\_idx in enumerate(top\_indices):

        reg\_clause = reg\_clauses[best\_idx]

        best\_score = float(similarities[best\_idx])

        should\_use\_llama = USE\_LLaMA and i < MAX\_LLaMA\_ANALYSIS

        llama\_response = generate\_llama\_analysis(cleaned\_control, reg\_clause["text"], best\_score, reg\_clause["regulation"]) if should\_use\_llama else "AI analysis not applied."

        parsed = parse\_llama\_response(llama\_response)

        result = {

            "control\_id": control\_clause.get("clause\_id", f"Control-{i+1}"),

            "control": cleaned\_control,

            "status": classify\_status(best\_score),

            "score": round(best\_score, 3),

            "matched\_clause": reg\_clause["text"],

            "regulation": reg\_clause["regulation"],

            "doc\_name": reg\_clause.get("doc\_name", "Unknown"),

            "page\_num": reg\_clause.get("page\_num", "—"),

            "section": reg\_clause.get("section", "—"),

            "overlap": parsed.get("overlap", "—"),

            "gap": parsed.get("gap", "—"),

            "reason": parsed.get("reason", "AI analysis not applied."),

            "rewrite": parsed.get("rewrite", "—"),

            "risk": parsed.get("risk", "—"),

            "fine": parsed.get("fine", "—")

        }

        results.append(result)

    return results

def process\_and\_match\_multiple\_docs(control\_clauses, regulation\_clauses, remove\_stopwords=True):

    reg\_clean = []

    for r in regulation\_clauses:

        reg\_clean.append({

            "text": clean\_text(r.get("text", "")),

            "regulation": r.get("regulation", "Unknown Regulation"),

            "doc\_name": r.get("doc\_name", "—"),

            "page\_num": r.get("page\_num", "—"),

            "section": r.get("section", "—")

        })

    reg\_embeddings = batch\_encode([r["text"] for r in reg\_clean])

    results = []

    with ThreadPoolExecutor() as executor:

        futures = [

            executor.submit(match\_single\_clause, i, clause, reg\_clean, reg\_embeddings)

            for i, clause in enumerate(control\_clauses)

        ]

        for future in futures:

            results.extend(future.result())

    return results

REPORT BUILDER

import io

import pandas as pd

from datetime import datetime

def clean\_field(val):

    if isinstance(val, str) and any(err in val for err in ["LLaMA Error", "Too Many Requests", "Payload Too Large"]):

        return "⚠️ Unable to generate reasoning (AI overload)"

    return val if val else "—"

def generate\_final\_csv\_report(matched\_controls, missing\_controls, chat\_history=None):

    output = io.BytesIO()

    def safe\_series(df, \*possible\_keys, default="—"):

        """Returns first existing column or a fallback series."""

        for key in possible\_keys:

            if key in df.columns:

                return df[key]

        return pd.Series([default] \* len(df))

    def preprocess(df\_raw, is\_missing=False):

        if not df\_raw:

            return pd.DataFrame(columns=[

                "Clause ID", "Control Clause", "Match Type", "Regulation", "Match Score (%)",

                "Overlap Terms", "Semantic Gap Analysis", "AI Reasoning",

                "Suggested Rewrite for Better Compliance", "Associated Risks", "Potential Fines"

            ])

        df = pd.DataFrame(df\_raw).copy()

        df["Clause ID"] = safe\_series(df, "Clause ID", "control\_id")

        df["Control Clause"] = safe\_series(df, "Control Clause", "control", "Missing Clause")

        df["Match Type"] = "Unmatched" if is\_missing else safe\_series(df, "Match Type", "status", default="—")

        df["Regulation"] = safe\_series(df, "Regulation", "regulation")

        df["Match Score (%)"] = safe\_series(df, "Score", "score", default=0.0)

        df["Overlap Terms"] = safe\_series(df, "Overlap Terms", "overlap")

        df["Semantic Gap Analysis"] = safe\_series(df, "Gap", "gap")

        # AI Details

        df["AI Reasoning"] = safe\_series(df, "Reasoning", "reason").apply(clean\_field)

        df["Suggested Rewrite for Better Compliance"] = safe\_series(df, "rewrite")

        df["Associated Risks"] = safe\_series(df, "risk")

        df["Potential Fines"] = safe\_series(df, "fine")

        return df[[

            "Clause ID",

            "Control Clause",

            "Match Type",

            "Regulation",

            "Match Score (%)",

            "Overlap Terms",

            "Semantic Gap Analysis",

            "AI Reasoning",

            "Suggested Rewrite for Better Compliance",

            "Associated Risks",

            "Potential Fines"

        ]]

    # Sheet 1: Full Compliance Summary

    df\_matched = preprocess(matched\_controls, is\_missing=False)

    df\_missing = preprocess(missing\_controls, is\_missing=True)

    df\_compliance = pd.concat([df\_matched, df\_missing], ignore\_index=True)

    # Sheet 2: Chatbot History

    chat\_rows = []

    if chat\_history:

        for msg in chat\_history:

            chat\_rows.append({

                "Role": msg.get("role", "system").title(),

                "Message": msg.get("content", "").replace("\n", " ").strip(),

                "Timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S")

            })

    df\_chat = pd.DataFrame(chat\_rows, columns=["Role", "Message", "Timestamp"])

    # Sheet 3: Rewritten Controls

    df\_rewrites = df\_matched[

        df\_matched["Suggested Rewrite for Better Compliance"] != "—"

    ][[

        "Clause ID",

        "Control Clause",

        "Suggested Rewrite for Better Compliance",

        "AI Reasoning",

        "Associated Risks",

        "Potential Fines"

    ]].copy()

    # Sheet 4: Cross-Regulation Mapping

    cross\_data = []

    for row in matched\_controls:

        cross\_data.append({

            "Clause ID": row.get("control\_id", "—"),

            "Appears In Regulation": row.get("regulation", "—"),

            "Overlap Terms": row.get("overlap", "—"),

            "Gaps Noted": row.get("gap", "—")

        })

    df\_cross = pd.DataFrame(cross\_data, columns=[

        "Clause ID", "Appears In Regulation", "Overlap Terms", "Gaps Noted"

    ])

    # Write to Excel

    with pd.ExcelWriter(output, engine="xlsxwriter") as writer:

        df\_compliance.to\_excel(writer, sheet\_name="Compliance Report", index=False)

        df\_chat.to\_excel(writer, sheet\_name="Chatbot History", index=False)

        df\_rewrites.to\_excel(writer, sheet\_name="Rewritten Controls", index=False)

        df\_cross.to\_excel(writer, sheet\_name="Cross-Document Analysis", index=False)

    output.seek(0)

    return output

DASHBOARD

import os

import sys

import logging

from pathlib import Path

import streamlit as st

# Add project root to sys.path

sys.path.append(str(Path(\_\_file\_\_).parent.parent.resolve()))

# App modules

from api.document\_parser import process\_uploaded\_file

from api.match\_engine import process\_and\_match\_multiple\_docs

from api.report\_builder import generate\_final\_csv\_report

from api.llama\_chat\_agent import ask\_llama, get\_flashcard\_prompts\_from\_context

# Logging

logging.basicConfig(level=logging.INFO)

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

# Streamlit Page Setup

st.set\_page\_config(page\_title="Compliance Intelligence Engine", layout="wide")

# 🧠 Session Init

def init\_session():

    st.session\_state.setdefault("chat\_history", [])

    st.session\_state.setdefault("processed\_data", None)

    st.session\_state.setdefault("context\_injected", False)

init\_session()

# 🎨 Dark Mode Styling

st.markdown("""

<style>

html, body, [data-testid="stAppViewContainer"] {

    background-color: #0e1117;

    color: #ffffff;

}

[data-testid="stSidebar"] {

    background-color: #1e1e1e;

}

.stTextInput > div > input {

    color: white; background-color: #262730;

}

.stButton > button {

    color: white; background-color: #2a2d3a;

}

thead tr th, tbody tr td {

    color: white !important;

}

</style>

""", unsafe\_allow\_html=True)

# 📂 Upload Section

st.sidebar.header("📂 Upload Documents")

control\_docs = st.sidebar.file\_uploader("Upload Company Controls", type=["pdf", "docx", "txt", "csv", "xlsx"], accept\_multiple\_files=True)

regulation\_docs = st.sidebar.file\_uploader("Upload Regulations", type=["pdf", "docx", "txt", "csv", "xlsx"], accept\_multiple\_files=True)

if not control\_docs or not regulation\_docs:

    st.sidebar.warning("Please upload both control and regulatory documents.")

    st.stop()

# 🔍 Run Matching

@st.cache\_data(show\_spinner=False)

def run\_matching():

    try:

        control\_clauses, regulation\_clauses = [], []

        for f in control\_docs:

            control\_clauses += process\_uploaded\_file(f)

        for f in regulation\_docs:

            clauses = process\_uploaded\_file(f)

            regulation\_clauses += [{"text": c["text"], "regulation": f.name} for c in clauses]

        results = process\_and\_match\_multiple\_docs(control\_clauses, regulation\_clauses)

        matched, missing = [], []

        for r in results:

            score = r.get("score", 0.0)

            if r["status"] == "Unmatched":

                missing.append({

                    "Missing Clause": r.get("control", "—"),

                    "Score": score,

                    "gap": r.get("gap", "—"),

                    "reason": r.get("reason", "—"),

                    "Regulation": r.get("regulation", "—"),

                    "rewrite": r.get("rewrite", "—"),

                    "risk": r.get("risk", "—"),

                    "fine": r.get("fine", "—")

                })

            else:

                matched.append({

                    "Clause ID": r.get("control\_id", "—"),

                    "Control Clause": r.get("control", "—"),

                    "Match Type": r.get("status", "—"),

                    "Score": score,

                    "Regulation": r.get("regulation", "—"),

                    "Overlap Terms": r.get("overlap", "—"),

                    "Gap": r.get("gap", "—"),

                    "Reasoning": r.get("reason", "—"),

                    "rewrite": r.get("rewrite", "—"),

                    "risk": r.get("risk", "—"),

                    "fine": r.get("fine", "—")

                })

        return {"matched": matched, "missing": missing}

    except Exception as e:

        logger.exception(e)

        st.error("❌ AI Matching failed.")

        return {"matched": [], "missing": []}

if st.sidebar.button("🔍 Run Compliance Matching"):

    with st.spinner("Running AI-based clause analysis..."):

        st.session\_state.processed\_data = run\_matching()

if st.sidebar.button("📥 Download CSV Report"):

    try:

        report = generate\_final\_csv\_report(

            st.session\_state.get("processed\_data", {}).get("matched", []),

            st.session\_state.get("processed\_data", {}).get("missing", []),

            chat\_history=st.session\_state.get("chat\_history", [])

        )

        st.sidebar.download\_button(

            label="📊 Export Compliance Report",

            data=report,

            file\_name="Compliance\_Report.xlsx",

            mime="application/vnd.openxmlformats-officedocument.spreadsheetml.sheet"

        )

    except Exception as e:

        logger.error(e)

        st.sidebar.error("❌ Report generation failed.")

# 🧠 Summary UI

if st.session\_state.get("processed\_data"):

    st.markdown("## 📊 Compliance Summary")

    top\_matches = st.session\_state["processed\_data"].get("matched", [])[:3]

    top\_misses = st.session\_state["processed\_data"].get("missing", [])[:2]

    for m in top\_matches:

        st.success(f"✅ `{m['Clause ID']}` matched in \*\*{m['Regulation']}\*\* (Score: {round(m['Score'], 2)})")

    for m in top\_misses:

        st.warning(f"⚠️ Missing: “{m['Missing Clause'][:60]}...” — Gap: \*\*{m['gap']}\*\*")

    # Flashcards

    st.markdown("### 🧠 Suggested Prompts")

    for prompt in get\_flashcard\_prompts\_from\_context():

        if st.button(prompt, key=f"prompt\_{prompt}"):

            st.session\_state.chat\_history.append({"role": "user", "content": prompt})

            try:

                with st.spinner("Thinking..."):

                    stream = ask\_llama(st.session\_state.chat\_history, stream=True)

                    full\_response = ""

                    for chunk in stream:

                        full\_response += chunk

                        st.write\_stream(iter([chunk]))

                st.session\_state.chat\_history.append({"role": "assistant", "content": full\_response})

            except Exception as e:

                st.error("❌ Chat failed.")

                logger.error(e)

# 💬 Chatbox

st.markdown("## 💬 Chat with Compliance Assistant")

with st.form("chat\_form", clear\_on\_submit=True):

    user\_input = st.text\_input("Ask about gaps, risks, rewrites or matches:")

    submitted = st.form\_submit\_button("📨 Send")

if submitted and user\_input:

    st.session\_state.chat\_history.append({"role": "user", "content": user\_input})

    try:

        with st.spinner("Analyzing..."):

            stream = ask\_llama(st.session\_state.chat\_history, stream=True)

            response = ""

            for chunk in stream:

                response += chunk

                st.write\_stream(iter([chunk]))

        st.session\_state.chat\_history.append({"role": "assistant", "content": response})

    except Exception as e:

        st.error("❌ Chat failed.")

        logger.error(e)

# 💾 Chat History

for msg in st.session\_state.chat\_history:

    with st.chat\_message(msg["role"]):

        st.markdown(msg["content"])

# ♻️ Reset

if st.sidebar.button("🧹 Reset All"):

    for key in ["chat\_history", "processed\_data", "context\_injected"]:

        st.session\_state[key] = [] if "history" in key else None

    st.rerun()

VERSION 3

DOCUMENT PARSER

# api/document\_parser.py

import os

import re

import time

import logging

from datetime import datetime

import pandas as pd

import nltk

from nltk.tokenize import sent\_tokenize

from nltk.corpus import stopwords

import fitz  # PyMuPDF

from docx import Document

# --- Logging Setup ---

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

logging.basicConfig(level=logging.INFO)

# --- NLTK Setup ---

try:

    nltk.data.find("tokenizers/punkt")

except LookupError:

    nltk.download("punkt")

try:

    nltk.data.find("corpora/stopwords")

except LookupError:

    nltk.download("stopwords")

stop\_words = set(stopwords.words("english"))

SUPPORTED\_EXTENSIONS = [".pdf", ".docx", ".txt", ".csv", ".xlsx", ".xls"]

KNOWN\_REGULATIONS = ["GDPR", "ISO27001", "RBI", "SEBI", "PDPB", "DPDP", "MSA"]

# --- Extractor Dispatcher ---

def extract\_text(file\_path):

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

    if ext not in SUPPORTED\_EXTENSIONS:

        raise ValueError(f"❌ Unsupported file format: {ext}")

    try:

        if ext == ".pdf":

            return extract\_pdf\_clauses(file\_path)

        elif ext == ".docx":

            return extract\_docx\_clauses(file\_path)

        elif ext == ".txt":

            return extract\_txt\_clauses(file\_path)

        elif ext == ".csv":

            return extract\_csv\_clauses(file\_path)

        elif ext in [".xlsx", ".xls"]:

            return extract\_excel\_clauses(file\_path)

    except Exception as e:

        raise Exception(f"⚠️ Error reading {file\_path}: {str(e)}")

# --- Format-Specific Extraction Functions ---

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

    clauses = []

    with fitz.open(file\_path) as doc:

        for page\_num, page in enumerate(doc, start=1):

            text = page.get\_text()

            for i, sent in enumerate(sent\_tokenize(text)):

                sent = sanitize\_clause(sent)

                if is\_valid\_clause(sent):

                    clauses.append(make\_clause\_dict(file\_path, sent, page\_num, f"Page {page\_num}", f"P{page\_num}-C{i+1}"))

    return clauses

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

    clauses = []

    doc = Document(file\_path)

    current\_section = ""

    para\_count = 0

    for para in doc.paragraphs:

        text = para.text.strip()

        if para.style.name.startswith("Heading"):

            current\_section = text

            continue

        if len(text) < 20:

            continue

        para\_count += 1

        sent = sanitize\_clause(text)

        if is\_valid\_clause(sent):

            clauses.append(make\_clause\_dict(file\_path, sent, None, current\_section or "Untitled", f"S{para\_count}"))

    return clauses

def extract\_txt\_clauses(file\_path):

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

        text = f.read()

    return split\_sentences\_into\_clauses(text, file\_path)

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

    df = pd.read\_csv(file\_path, encoding="utf-8", on\_bad\_lines="skip")

    clauses = []

    for i, row in df.iterrows():

        joined = " ".join(map(str, row.dropna())).strip()

        joined = sanitize\_clause(joined)

        if is\_valid\_clause(joined):

            clauses.append(make\_clause\_dict(file\_path, joined, None, "Row", f"R{i+1}"))

    return clauses

def extract\_excel\_clauses(file\_path):

    df = pd.read\_excel(file\_path)

    clauses = []

    for i, row in df.iterrows():

        joined = " ".join(map(str, row.dropna())).strip()

        joined = sanitize\_clause(joined)

        if is\_valid\_clause(joined):

            clauses.append(make\_clause\_dict(file\_path, joined, None, "Sheet", f"XL{i+1}"))

    return clauses

def split\_sentences\_into\_clauses(text, file\_path):

    clauses = []

    for i, sent in enumerate(sent\_tokenize(text)):

        sent = sanitize\_clause(sent)

        if is\_valid\_clause(sent):

            clauses.append(make\_clause\_dict(file\_path, sent, None, "Text File", f"T{i+1}"))

    return clauses

# --- Shared Helpers ---

def make\_clause\_dict(file\_path, text, page\_num, section, suffix):

    base = os.path.splitext(os.path.basename(file\_path))[0]

    return {

        "clause\_id": f"{base}-{suffix}",

        "text": text,

        "doc\_name": os.path.basename(file\_path),

        "page\_num": page\_num,

        "section": section,

        "source\_type": infer\_source\_type(file\_path)

    }

def sanitize\_clause(clause):

    s = re.sub(r"[^A-Za-z0-9\s,.()\-–/]", "", clause.strip())

    s = re.sub(r"\s+", " ", s)

    return s

def is\_valid\_clause(text):

    if any(skip in text.lower() for skip in ["table of contents", "annexure", "appendix"]):

        return False

    return len(text) >= 20 and any(c.isalpha() for c in text)

def infer\_source\_type(file\_path):

    fname = os.path.basename(file\_path).upper()

    if "CONTROL" in fname or "POLICY" in fname:

        return "Control"

    elif any(reg in fname for reg in KNOWN\_REGULATIONS):

        return "Regulation"

    else:

        return "Unknown"

# --- Save Upload ---

def save\_uploaded\_file(uploaded\_file, save\_dir="data/uploads"):

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

    safe\_filename = re.sub(r"[^A-Za-z0-9\_\-\.]", "\_", uploaded\_file.name)

    file\_path = os.path.join(save\_dir, safe\_filename)

    with open(file\_path, "wb") as f:

        f.write(uploaded\_file.getbuffer())

    return file\_path

def save\_text\_and\_metadata(text, original\_filename, save\_dir="data/texts"):

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

    base = os.path.splitext(original\_filename)[0]

    safe\_base = re.sub(r"[^A-Za-z0-9\_\-]", "\_", base)

    text\_path = os.path.join(save\_dir, f"{safe\_base}.txt")

    meta\_path = os.path.join(save\_dir, f"{safe\_base}\_meta.txt")

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

        f.write(text)

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

        f.write(f"📂 File: {original\_filename}\n")

        f.write(f"📦 Size (KB): {len(text.encode('utf-8')) // 1024}\n")

        f.write(f"⏱️ Processed on: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n")

    return text\_path, meta\_path

# --- Full Pipeline ---

def process\_uploaded\_file(uploaded\_file, save\_dir="data/uploads"):

    start = time.time()

    file\_path = save\_uploaded\_file(uploaded\_file, save\_dir)

    logger.info(f"[✓] Saved: {file\_path}")

    clauses = extract\_text(file\_path)

    flat\_text = "\n".join([c["text"] for c in clauses])

    save\_text\_and\_metadata(flat\_text, uploaded\_file.name, save\_dir="data/texts")

    logger.info(f"[✓] Extracted {len(clauses)} clauses from {uploaded\_file.name} in {time.time() - start:.2f}s")

    return clauses

LLAMA CHAT AGENT

# api/llama\_chat\_agent.py

# api/llama\_chat\_agent.py

import re

import os

import streamlit as st

from typing import List, Dict, Generator, Union

from dotenv import load\_dotenv

from openai import OpenAI

# Load API key from .env

load\_dotenv()

api\_key = os.getenv("GROQ\_API\_KEY")

if not api\_key:

    raise ValueError("🚨 GROQ\_API\_KEY not found. Check your .env file.")

client = OpenAI(

    base\_url="https://api.groq.com/openai/v1",

    api\_key=api\_key

)

# --- Utility: Token Counting ---

def approximate\_token\_count(text: str) -> int:

    return len(re.findall(r"\w+|[^\w\s]", text, re.UNICODE))

def dynamic\_max\_tokens(messages: List[Dict], model\_max\_tokens: int = 8192) -> int:

    used = sum(approximate\_token\_count(m.get("content", "")) for m in messages)

    return max(model\_max\_tokens - used - 150, 150)

# --- Match Summary Context Injection ---

def get\_memory\_context() -> Union[Dict, None]:

    if not st.session\_state.get("processed\_data"):

        return None

    matched = st.session\_state["processed\_data"].get("matched", [])[:5]

    missing = st.session\_state["processed\_data"].get("missing", [])[:3]

    context\_lines = []

    if matched:

        context\_lines.append("✅ Top Matched Clauses:")

        for item in matched:

            cid = item.get("control\_id", "Clause")

            reg = item.get("regulation", "Regulation")

            score = item.get("score", "—")

            clause\_text = item.get("control", "")[:80]

            context\_lines.append(f"- `{cid}` in \*\*{reg}\*\* (Score: {score}): \"{clause\_text}...\"")

    if missing:

        context\_lines.append("❌ Missing Clauses:")

        for item in missing:

            clause\_text = item.get("control", "")[:70]

            gap = item.get("gap", "—")

            context\_lines.append(f"- \"{clause\_text}...\"  (Gap: {gap})")

    return {

        "role": "system",

        "content": (

            "You are a legal compliance AI assistant. In Audit Mode, only use uploaded clause data.\n"

            "Always cite specific clause IDs and their regulation sources.\n\n"

            "📄 Clause Summary:\n" + "\n".join(context\_lines)

        )

    }

def inject\_context\_once(chat\_history: List[Dict]) -> List[Dict]:

    if not st.session\_state.get("context\_injected", False):

        context = get\_memory\_context()

        if context:

            st.session\_state.context\_injected = True

            return [context] + [m for m in chat\_history if m["role"] != "system"]

    return chat\_history

# --- Chat Function ---

def ask\_llama(

    messages: Union[str, List[Dict]],

    model: str = "llama3-8b-8192",  # use 8b model for token safety

    temperature: float = 0.2,

    stream: bool = False,

    audit\_mode: bool = True

) -> Union[str, Generator[str, None, None]]:

    if isinstance(messages, str):

        messages = [{"role": "user", "content": messages}]

    messages = inject\_context\_once(messages)

    if audit\_mode:

        system\_msg = {

            "role": "system",

            "content": (

                "You are in AUDIT MODE.\n"

                "Only use uploaded clause content (matched or missing).\n"

                "If unsure, respond: \"I cannot verify this based on uploaded documents.\"\n"

                "Cite clause IDs and regulation names always. Do not assume or hallucinate."

            )

        }

        messages.insert(0, system\_msg)

    max\_tokens = dynamic\_max\_tokens(messages)

    params = {

        "model": model,

        "messages": messages,

        "max\_tokens": max\_tokens,

        "temperature": temperature,

        "stream": stream

    }

    try:

        if stream:

            response = client.chat.completions.create(\*\*params)

            return (chunk.choices[0].delta.content or "" for chunk in response)

        else:

            response = client.chat.completions.create(\*\*params)

            reply = response.choices[0].message.content.strip()

            if response.choices[0].finish\_reason != "stop":

                reply += "\n\n⚠️ Truncated due to token limit."

            return reply

    except Exception as e:

        return (

            "⚠️ The system was unable to analyze this fully due to token limit or API failure.\n"

            f"Error: {str(e)}\n"

            "Please try a shorter query or re-run matching."

        )

# --- Flashcard Generator ---

def get\_flashcard\_prompts\_from\_context() -> List[str]:

    prompts = []

    data = st.session\_state.get("processed\_data")

    if not data:

        return prompts

    matched = data.get("matched", [])

    missing = data.get("missing", [])

    if matched:

        top = matched[0]

        cid = top.get("control\_id", "Control")

        reg = top.get("regulation", "Regulation")

        prompts.append(f"🔍 How does \*\*{cid}\*\* align with {reg}?")

        prompts.append(f"💡 Can \*\*{cid}\*\* be improved to meet full compliance?")

    if missing:

        top = missing[0]

        gap = top.get("gap", "—")

        text = top.get("control", "")[:50]

        prompts.append(f"⚠️ Why is this clause marked missing? “{text}...”")

        prompts.append(f"📘 Suggest a new control to cover: {gap}")

    prompts.append("🧠 What's the overall risk profile based on current matches?")

    return prompts

MATCH ENGINE

# api/match\_engine.py

import os

import re

import numpy as np

from nltk.corpus import stopwords

from sentence\_transformers import SentenceTransformer, util

from concurrent.futures import ThreadPoolExecutor

import streamlit as st

import httpx

import json

# --- Config ---

model = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")

THRESH\_STRONG = 0.75

THRESH\_PARTIAL = 0.5

THRESH\_WEAK = 0.25

TOP\_K = 1  # How many top matches per control clause

STOPWORDS = set(stopwords.words("english"))

USE\_LLaMA = True

MAX\_LLaMA\_ANALYSIS = st.session\_state.get("max\_llama\_clauses", 3)

GROQ\_API\_KEY = os.getenv("GROQ\_API\_KEY")

LLAMA\_MODEL = "llama3-70b-8192"

GROQ\_URL = "https://api.groq.com/openai/v1/chat/completions"

@st.cache\_data(show\_spinner=False)

def batch\_encode(texts):

    return model.encode(texts, convert\_to\_tensor=True, show\_progress\_bar=False)

def clean\_text(text):

    return re.sub(r'\s+', ' ', text.strip())

def extract\_words(text, remove\_stopwords=True):

    words = set(re.findall(r'\w+', text.lower()))

    return words - STOPWORDS if remove\_stopwords else words

def classify\_status(score):

    if score >= THRESH\_STRONG:

        return "Strong Match"

    elif score >= THRESH\_PARTIAL:

        return "Partial Match"

    elif score >= THRESH\_WEAK:

        return "Weak Match"

    else:

        return "Unmatched"

def parse\_llama\_response(response\_text):

    result = {

        "rewrite": "—",

        "overlap": "—",

        "gap": "—",

        "reason": "—",

        "risk": "—",

        "fine": "—"

    }

    try:

        # Simple line-based breakdown

        lines = response\_text.split("\n")

        for line in lines:

            l = line.lower()

            if "overlap" in l:

                result["overlap"] = line.split(":")[-1].strip()

            elif "gap" in l or "missing" in l:

                result["gap"] = line.split(":")[-1].strip()

            elif "rewrite" in l:

                result["rewrite"] = line.split(":", 1)[-1].strip()

            elif "risk" in l:

                result["risk"] = line.split(":", 1)[-1].strip()

            elif "fine" in l:

                result["fine"] = line.split(":", 1)[-1].strip()

            elif "classify" in l or "match" in l:

                result["reason"] = line.strip()

    except Exception:

        result["reason"] = response\_text.strip()

    return result

def generate\_llama\_analysis(control, regulation, score, reg\_name):

    prompt = f"""

Control Clause:

\"\"\"{control}\"\"\"

Regulation Clause from {reg\_name}:

\"\"\"{regulation}\"\"\"

Match score: {round(score, 3)}

You are a compliance analyst. Analyze the above in detail:

1. Classify the match: Strong, Partial, Weak, Unmatched

2. 🔗 Overlap

3. 📉 Gaps

4. 🧠 Rewrite (if needed)

5. ⚠️ Non-Compliance Risk

6. 💸 Estimated Fine (Low, Medium, High + reason)

Provide your answer in plain text format.

"""

    headers = {

        "Authorization": f"Bearer {GROQ\_API\_KEY}",

        "Content-Type": "application/json"

    }

    payload = {

        "model": LLAMA\_MODEL,

        "temperature": 0.4,

        "max\_tokens": 400,

        "messages": [{"role": "user", "content": prompt}]

    }

    try:

        response = httpx.post(GROQ\_URL, json=payload, headers=headers, timeout=45.0)

        response.raise\_for\_status()

        return response.json()["choices"][0]["message"]["content"].strip()

    except Exception as e:

        return f"[LLaMA Error: {str(e)}]"

def match\_single\_clause(i, control\_clause, reg\_clauses, reg\_embeddings):

    control\_text = control\_clause.get("text", "").strip()

    if not control\_text:

        return [{

            "control\_id": control\_clause.get("clause\_id", f"Control-{i+1}"),

            "control": "[EMPTY]",

            "status": "Unmatched",

            "score": 0.0,

            "matched\_clause": "—",

            "regulation": "—",

            "doc\_name": "—",

            "page\_num": "—",

            "section": "—",

            "overlap": "—",

            "gap": "Clause was empty",

            "reason": "—",

            "rewrite": "—",

            "risk": "—",

            "fine": "—"

        }]

    cleaned\_control = clean\_text(control\_text)

    control\_embedding = model.encode(cleaned\_control, convert\_to\_tensor=True)

    similarities = util.cos\_sim(control\_embedding, reg\_embeddings)[0].cpu().numpy()

    top\_indices = similarities.argsort()[-TOP\_K:][::-1]

    results = []

    for j, best\_idx in enumerate(top\_indices):

        reg\_clause = reg\_clauses[best\_idx]

        best\_score = float(similarities[best\_idx])

        should\_use\_llama = USE\_LLaMA and i < MAX\_LLaMA\_ANALYSIS

        llama\_response = generate\_llama\_analysis(cleaned\_control, reg\_clause["text"], best\_score, reg\_clause["regulation"]) if should\_use\_llama else "AI analysis not applied."

        parsed = parse\_llama\_response(llama\_response)

        result = {

            "control\_id": control\_clause.get("clause\_id", f"Control-{i+1}"),

            "control": cleaned\_control,

            "status": classify\_status(best\_score),

            "score": round(best\_score, 3),

            "matched\_clause": reg\_clause["text"],

            "regulation": reg\_clause["regulation"],

            "doc\_name": reg\_clause.get("doc\_name", "Unknown"),

            "page\_num": reg\_clause.get("page\_num", "—"),

            "section": reg\_clause.get("section", "—"),

            "overlap": parsed.get("overlap", "—"),

            "gap": parsed.get("gap", "—"),

            "reason": parsed.get("reason", "AI analysis not applied."),

            "rewrite": parsed.get("rewrite", "—"),

            "risk": parsed.get("risk", "—"),

            "fine": parsed.get("fine", "—")

        }

        results.append(result)

    return results

def process\_and\_match\_multiple\_docs(control\_clauses, regulation\_clauses, remove\_stopwords=True):

    reg\_clean = []

    for r in regulation\_clauses:

        reg\_clean.append({

            "text": clean\_text(r.get("text", "")),

            "regulation": r.get("regulation", "Unknown Regulation"),

            "doc\_name": r.get("doc\_name", "—"),

            "page\_num": r.get("page\_num", "—"),

            "section": r.get("section", "—")

        })

    reg\_embeddings = batch\_encode([r["text"] for r in reg\_clean])

    results = []

    with ThreadPoolExecutor() as executor:

        futures = [

            executor.submit(match\_single\_clause, i, clause, reg\_clean, reg\_embeddings)

            for i, clause in enumerate(control\_clauses)

        ]

        for future in futures:

            results.extend(future.result())

    return results

REPORT BUILDER

import io

import pandas as pd

from datetime import datetime

def clean\_field(val):

    if isinstance(val, str) and any(err in val for err in ["LLaMA Error", "Too Many Requests", "Payload Too Large"]):

        return "⚠️ Unable to generate reasoning (AI overload)"

    return val if val else "—"

def generate\_final\_csv\_report(matched\_controls, missing\_controls, chat\_history=None, audit\_mode=True):

    output = io.BytesIO()

    def safe\_series(df, \*possible\_keys, default="—"):

        for key in possible\_keys:

            if key in df.columns:

                return df[key]

        return pd.Series([default] \* len(df))

    def preprocess(df\_raw, is\_missing=False):

        if not df\_raw:

            return pd.DataFrame(columns=[

                "Clause ID", "Control Clause", "Match Type", "Regulation", "Match Score (%)",

                "Overlap Terms", "Semantic Gap Analysis", "AI Reasoning",

                "Suggested Rewrite for Better Compliance", "Associated Risks", "Potential Fines"

            ])

        df = pd.DataFrame(df\_raw).copy()

        df["Clause ID"] = safe\_series(df, "Clause ID", "control\_id")

        df["Control Clause"] = safe\_series(df, "Control Clause", "control", "Missing Clause")

        df["Match Type"] = "Unmatched" if is\_missing else safe\_series(df, "Match Type", "status", default="—")

        df["Regulation"] = safe\_series(df, "Regulation", "regulation")

        df["Match Score (%)"] = safe\_series(df, "Score", "score", default=0.0)

        df["Overlap Terms"] = safe\_series(df, "Overlap Terms", "overlap")

        df["Semantic Gap Analysis"] = safe\_series(df, "Gap", "gap")

        df["AI Reasoning"] = safe\_series(df, "Reasoning", "reason").apply(clean\_field)

        df["Suggested Rewrite for Better Compliance"] = safe\_series(df, "rewrite")

        df["Associated Risks"] = safe\_series(df, "risk")

        df["Potential Fines"] = safe\_series(df, "fine")

        return df[[

            "Clause ID",

            "Control Clause",

            "Match Type",

            "Regulation",

            "Match Score (%)",

            "Overlap Terms",

            "Semantic Gap Analysis",

            "AI Reasoning",

            "Suggested Rewrite for Better Compliance",

            "Associated Risks",

            "Potential Fines"

        ]]

    # --- Sheet 1: Full Summary ---

    df\_matched = preprocess(matched\_controls, is\_missing=False)

    df\_missing = preprocess(missing\_controls, is\_missing=True)

    df\_compliance = pd.concat([df\_matched, df\_missing], ignore\_index=True)

    # --- Sheet 2: Chat History ---

    chat\_rows = []

    if chat\_history:

        for msg in chat\_history:

            chat\_rows.append({

                "Role": msg.get("role", "system").title(),

                "Message": msg.get("content", "").replace("\n", " ").strip(),

                "Timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S")

            })

    df\_chat = pd.DataFrame(chat\_rows, columns=["Role", "Message", "Timestamp"])

    # --- Sheet 3: Rewrites ---

    df\_rewrites = df\_matched[

        df\_matched["Suggested Rewrite for Better Compliance"] != "—"

    ][[

        "Clause ID", "Control Clause", "Suggested Rewrite for Better Compliance",

        "AI Reasoning", "Associated Risks", "Potential Fines"

    ]].copy()

    # --- Sheet 4: Cross Regulation ---

    cross\_data = []

    for row in matched\_controls:

        cross\_data.append({

            "Clause ID": row.get("control\_id", "—"),

            "Appears In Regulation": row.get("regulation", "—"),

            "Overlap Terms": row.get("overlap", "—"),

            "Gaps Noted": row.get("gap", "—")

        })

    df\_cross = pd.DataFrame(cross\_data, columns=[

        "Clause ID", "Appears In Regulation", "Overlap Terms", "Gaps Noted"

    ])

    # --- Sheet 5: Audit Mode Info ---

    session\_info = pd.DataFrame([

        {"Field": "Audit Mode", "Value": "ON" if audit\_mode else "OFF"},

        {"Field": "Generated On", "Value": datetime.now().strftime("%Y-%m-%d %H:%M:%S")},

        {"Field": "Matched Clauses", "Value": len(df\_matched)},

        {"Field": "Missing Clauses", "Value": len(df\_missing)}

    ])

    # --- Write to Excel ---

    with pd.ExcelWriter(output, engine="xlsxwriter") as writer:

        df\_compliance.to\_excel(writer, sheet\_name="Compliance Report", index=False)

        df\_chat.to\_excel(writer, sheet\_name="Chatbot History", index=False)

        df\_rewrites.to\_excel(writer, sheet\_name="Rewritten Controls", index=False)

        df\_cross.to\_excel(writer, sheet\_name="Cross-Document Analysis", index=False)

        session\_info.to\_excel(writer, sheet\_name="Session Info", index=False)

    output.seek(0)

    return output

DASHBOARD.PY

import os

import sys

import logging

from pathlib import Path

import streamlit as st

# Add project root to sys.path

sys.path.append(str(Path(\_\_file\_\_).parent.parent.resolve()))

# App modules

from api.document\_parser import process\_uploaded\_file

from api.match\_engine import process\_and\_match\_multiple\_docs

from api.report\_builder import generate\_final\_csv\_report

from api.llama\_chat\_agent import ask\_llama, get\_flashcard\_prompts\_from\_context

# Logging

logging.basicConfig(level=logging.INFO)

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

# Streamlit Page Setup

st.set\_page\_config(page\_title="Compliance Intelligence Engine", layout="wide")

# 🧠 Session Init

def init\_session():

    st.session\_state.setdefault("chat\_history", [])

    st.session\_state.setdefault("processed\_data", None)

    st.session\_state.setdefault("context\_injected", False)

init\_session()

# 🎨 Dark Mode Styling

st.markdown("""

<style>

html, body, [data-testid="stAppViewContainer"] {

    background-color: #0e1117;

    color: #ffffff;

}

[data-testid="stSidebar"] {

    background-color: #1e1e1e;

}

.stTextInput > div > input {

    color: white; background-color: #262730;

}

.stButton > button {

    color: white; background-color: #2a2d3a;

}

thead tr th, tbody tr td {

    color: white !important;

}

</style>

""", unsafe\_allow\_html=True)

# 📂 Upload Section

st.sidebar.header("📂 Upload Documents")

control\_docs = st.sidebar.file\_uploader("Upload Company Controls", type=["pdf", "docx", "txt", "csv", "xlsx"], accept\_multiple\_files=True)

regulation\_docs = st.sidebar.file\_uploader("Upload Regulations", type=["pdf", "docx", "txt", "csv", "xlsx"], accept\_multiple\_files=True)

# 🛡️ Audit Mode Toggle

st.sidebar.markdown("---")

audit\_mode\_enabled = st.sidebar.toggle("🛡️ Enable Audit Mode", value=True, help="Strict mode: Only use uploaded content, no assumptions.")

if not control\_docs or not regulation\_docs:

    st.sidebar.warning("Please upload both control and regulatory documents.")

    st.stop()

# 🔍 Run Matching

@st.cache\_data(show\_spinner=False)

def run\_matching():

    try:

        control\_clauses, regulation\_clauses = [], []

        for f in control\_docs:

            control\_clauses += process\_uploaded\_file(f)

        for f in regulation\_docs:

            clauses = process\_uploaded\_file(f)

            regulation\_clauses += [{"text": c["text"], "regulation": f.name} for c in clauses]

        results = process\_and\_match\_multiple\_docs(control\_clauses, regulation\_clauses)

        matched, missing = [], []

        for r in results:

            score = r.get("score", 0.0)

            if r["status"] == "Unmatched":

                missing.append({

                    "Missing Clause": r.get("control", "—"),

                    "Score": score,

                    "gap": r.get("gap", "—"),

                    "reason": r.get("reason", "—"),

                    "Regulation": r.get("regulation", "—"),

                    "rewrite": r.get("rewrite", "—"),

                    "risk": r.get("risk", "—"),

                    "fine": r.get("fine", "—")

                })

            else:

                matched.append({

                    "Clause ID": r.get("control\_id", "—"),

                    "Control Clause": r.get("control", "—"),

                    "Match Type": r.get("status", "—"),

                    "Score": score,

                    "Regulation": r.get("regulation", "—"),

                    "Overlap Terms": r.get("overlap", "—"),

                    "Gap": r.get("gap", "—"),

                    "Reasoning": r.get("reason", "—"),

                    "rewrite": r.get("rewrite", "—"),

                    "risk": r.get("risk", "—"),

                    "fine": r.get("fine", "—")

                })

        return {"matched": matched, "missing": missing}

    except Exception as e:

        logger.exception(e)

        st.error("❌ AI Matching failed.")

        return {"matched": [], "missing": []}

if st.sidebar.button("🔍 Run Compliance Matching"):

    with st.spinner("Running AI-based clause analysis..."):

        st.session\_state.processed\_data = run\_matching()

if st.sidebar.button("📥 Download CSV Report"):

    try:

        report = generate\_final\_csv\_report(

            st.session\_state.get("processed\_data", {}).get("matched", []),

            st.session\_state.get("processed\_data", {}).get("missing", []),

            chat\_history=st.session\_state.get("chat\_history", [])

        )

        st.sidebar.download\_button(

            label="📊 Export Compliance Report",

            data=report,

            file\_name="Compliance\_Report.xlsx",

            mime="application/vnd.openxmlformats-officedocument.spreadsheetml.sheet"

        )

    except Exception as e:

        logger.error(e)

        st.sidebar.error("❌ Report generation failed.")

# 🧠 Summary UI

if st.session\_state.get("processed\_data"):

    st.markdown("## 📊 Compliance Summary")

    top\_matches = st.session\_state["processed\_data"].get("matched", [])[:3]

    top\_misses = st.session\_state["processed\_data"].get("missing", [])[:2]

    for m in top\_matches:

        st.success(f"✅ `{m['Clause ID']}` matched in \*\*{m['Regulation']}\*\* (Score: {round(m['Score'], 2)})")

    for m in top\_misses:

        st.warning(f"⚠️ Missing: “{m['Missing Clause'][:60]}...” — Gap: \*\*{m['gap']}\*\*")

    # Flashcards

    st.markdown("### 🧠 Suggested Prompts")

    for prompt in get\_flashcard\_prompts\_from\_context():

        if st.button(prompt, key=f"prompt\_{prompt}"):

            st.session\_state.chat\_history.append({"role": "user", "content": prompt})

            try:

                with st.spinner("Thinking..."):

                    stream = ask\_llama(st.session\_state.chat\_history, stream=True, audit\_mode=audit\_mode\_enabled)

                    full\_response = ""

                    for chunk in stream:

                        full\_response += chunk

                        st.write\_stream(iter([chunk]))

                st.session\_state.chat\_history.append({"role": "assistant", "content": full\_response})

            except Exception as e:

                st.error("❌ Chat failed.")

                logger.error(e)

# 💬 Chatbox

st.markdown("## 💬 Chat with Compliance Assistant")

with st.form("chat\_form", clear\_on\_submit=True):

    user\_input = st.text\_input("Ask about gaps, risks, rewrites or matches:")

    submitted = st.form\_submit\_button("📨 Send")

if submitted and user\_input:

    st.session\_state.chat\_history.append({"role": "user", "content": user\_input})

    try:

        with st.spinner("Analyzing..."):

            stream = ask\_llama(st.session\_state.chat\_history, stream=True, audit\_mode=audit\_mode\_enabled)

            response = ""

            for chunk in stream:

                response += chunk

                st.write\_stream(iter([chunk]))

        st.session\_state.chat\_history.append({"role": "assistant", "content": response})

    except Exception as e:

        st.error("❌ Chat failed.")

        logger.error(e)

# 💾 Chat History

for msg in st.session\_state.chat\_history:

    with st.chat\_message(msg["role"]):

        st.markdown(msg["content"])

# ♻️ Reset

if st.sidebar.button("🧹 Reset All"):

    for key in ["chat\_history", "processed\_data", "context\_injected"]:

        st.session\_state[key] = [] if "history" in key else None

    st.rerun()

VERSION 4 WITH AUDIT MOD : IN FILE 1.2

# api/document\_parser.py

import os

import re

import time

import logging

from datetime import datetime

import pandas as pd

import nltk

from nltk.tokenize import sent\_tokenize

from nltk.corpus import stopwords

import fitz  # PyMuPDF

from docx import Document

# --- Logging Setup ---

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

logging.basicConfig(level=logging.INFO)

# --- NLTK Setup ---

try:

    nltk.data.find("tokenizers/punkt")

except LookupError:

    nltk.download("punkt")

try:

    nltk.data.find("corpora/stopwords")

except LookupError:

    nltk.download("stopwords")

stop\_words = set(stopwords.words("english"))

SUPPORTED\_EXTENSIONS = [".pdf", ".docx", ".txt", ".csv", ".xlsx", ".xls"]

KNOWN\_REGULATIONS = ["GDPR", "ISO27001", "RBI", "SEBI", "PDPB", "DPDP", "MSA"]

# --- Extractor Dispatcher ---

def extract\_text(file\_path):

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

    if ext not in SUPPORTED\_EXTENSIONS:

        raise ValueError(f"❌ Unsupported file format: {ext}")

    try:

        if ext == ".pdf":

            return extract\_pdf\_clauses(file\_path)

        elif ext == ".docx":

            return extract\_docx\_clauses(file\_path)

        elif ext == ".txt":

            return extract\_txt\_clauses(file\_path)

        elif ext == ".csv":

            return extract\_csv\_clauses(file\_path)

        elif ext in [".xlsx", ".xls"]:

            return extract\_excel\_clauses(file\_path)

    except Exception as e:

        raise Exception(f"⚠️ Error reading {file\_path}: {str(e)}")

# --- Format-Specific Extraction Functions ---

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

    clauses = []

    with fitz.open(file\_path) as doc:

        for page\_num, page in enumerate(doc, start=1):

            text = page.get\_text()

            for i, sent in enumerate(sent\_tokenize(text)):

                sent = sanitize\_clause(sent)

                if is\_valid\_clause(sent):

                    clauses.append(make\_clause\_dict(file\_path, sent, page\_num, f"Page {page\_num}", f"P{page\_num}-C{i+1}"))

    return clauses

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

    clauses = []

    doc = Document(file\_path)

    current\_section = ""

    para\_count = 0

    for para in doc.paragraphs:

        text = para.text.strip()

        if para.style.name.startswith("Heading"):

            current\_section = text

            continue

        if len(text) < 20:

            continue

        para\_count += 1

        sent = sanitize\_clause(text)

        if is\_valid\_clause(sent):

            clauses.append(make\_clause\_dict(file\_path, sent, None, current\_section or "Untitled", f"S{para\_count}"))

    return clauses

def extract\_txt\_clauses(file\_path):

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

        text = f.read()

    return split\_sentences\_into\_clauses(text, file\_path)

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

    df = pd.read\_csv(file\_path, encoding="utf-8", on\_bad\_lines="skip")

    clauses = []

    for i, row in df.iterrows():

        joined = " ".join(map(str, row.dropna())).strip()

        joined = sanitize\_clause(joined)

        if is\_valid\_clause(joined):

            clauses.append(make\_clause\_dict(file\_path, joined, None, "Row", f"R{i+1}"))

    return clauses

def extract\_excel\_clauses(file\_path):

    df = pd.read\_excel(file\_path)

    clauses = []

    for i, row in df.iterrows():

        joined = " ".join(map(str, row.dropna())).strip()

        joined = sanitize\_clause(joined)

        if is\_valid\_clause(joined):

            clauses.append(make\_clause\_dict(file\_path, joined, None, "Sheet", f"XL{i+1}"))

    return clauses

def split\_sentences\_into\_clauses(text, file\_path):

    clauses = []

    for i, sent in enumerate(sent\_tokenize(text)):

        sent = sanitize\_clause(sent)

        if is\_valid\_clause(sent):

            clauses.append(make\_clause\_dict(file\_path, sent, None, "Text File", f"T{i+1}"))

    return clauses

# --- Shared Helpers ---

def make\_clause\_dict(file\_path, text, page\_num, section, suffix):

    base = os.path.splitext(os.path.basename(file\_path))[0]

    return {

        "clause\_id": f"{base}-{suffix}",

        "text": text,

        "doc\_name": os.path.basename(file\_path),

        "page\_num": page\_num,

        "section": section,

        "source\_type": infer\_source\_type(file\_path)

    }

def sanitize\_clause(clause):

    s = re.sub(r"[^A-Za-z0-9\s,.()\-–/]", "", clause.strip())

    s = re.sub(r"\s+", " ", s)

    return s

def is\_valid\_clause(text):

    if any(skip in text.lower() for skip in ["table of contents", "annexure", "appendix"]):

        return False

    return len(text) >= 20 and any(c.isalpha() for c in text)

def infer\_source\_type(file\_path):

    fname = os.path.basename(file\_path).upper()

    if "CONTROL" in fname or "POLICY" in fname:

        return "Control"

    elif any(reg in fname for reg in KNOWN\_REGULATIONS):

        return "Regulation"

    else:

        return "Unknown"

# --- Save Upload ---

def save\_uploaded\_file(uploaded\_file, save\_dir="data/uploads"):

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

    safe\_filename = re.sub(r"[^A-Za-z0-9\_\-\.]", "\_", uploaded\_file.name)

    file\_path = os.path.join(save\_dir, safe\_filename)

    with open(file\_path, "wb") as f:

        f.write(uploaded\_file.getbuffer())

    return file\_path

def save\_text\_and\_metadata(text, original\_filename, save\_dir="data/texts"):

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

    base = os.path.splitext(original\_filename)[0]

    safe\_base = re.sub(r"[^A-Za-z0-9\_\-]", "\_", base)

    text\_path = os.path.join(save\_dir, f"{safe\_base}.txt")

    meta\_path = os.path.join(save\_dir, f"{safe\_base}\_meta.txt")

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

        f.write(text)

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

        f.write(f"📂 File: {original\_filename}\n")

        f.write(f"📦 Size (KB): {len(text.encode('utf-8')) // 1024}\n")

        f.write(f"⏱️ Processed on: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n")

    return text\_path, meta\_path

# --- Full Pipeline ---

def process\_uploaded\_file(uploaded\_file, save\_dir="data/uploads"):

    start = time.time()

    file\_path = save\_uploaded\_file(uploaded\_file, save\_dir)

    logger.info(f"[✓] Saved: {file\_path}")

    clauses = extract\_text(file\_path)

    flat\_text = "\n".join([c["text"] for c in clauses])

    save\_text\_and\_metadata(flat\_text, uploaded\_file.name, save\_dir="data/texts")

    logger.info(f"[✓] Extracted {len(clauses)} clauses from {uploaded\_file.name} in {time.time() - start:.2f}s")

    return clauses