**CCC Compliance Check Framework**

**1. Requirements**

1. **Detect and filter harmful content**
   * Violence, hate, sexual content, self-harm
2. **Enforce copyright protection**
   * Refuse requests for verbatim song lyrics, recipes, news, etc.
3. **Ground all QA/RAG outputs**
   * Only use facts from provided source documents
4. **Ground summarization outputs**
   * No extra or inferred information; focus on main points
5. **Add auditing and refusal logic**
   * Grade every request on a 0–1 scale; refuse “high‐risk” illicit behavior
6. **Build an API**
   * FastAPI endpoint that runs an LLM against these rules and logs everything
7. **Test and evaluate**
   * Generate domain-specific test cases (SDLC & Networking)
   * Capture ground truth and compute performance metrics

**2. Safety & Grounding Clauses**

We distilled the various “safety system messages” into four sets of clauses, encoded in **system.md**:

1. **Harm Prevention**
   * **H1:** No physically or emotionally harmful content
   * **H2:** No hateful, racist, sexist, lewd, or violent content
   * **H3 (Illicit Behavior):** No instructions for illegal or malicious activities
2. **Protected Material**
   * **P1:** Refuse requests for full copyrighted text; offer summary only
3. **Groundedness & RAG Safety**
   * **G1–G8:** Always base answers on provided documents, cite facts, avoid assumptions
4. **Summarization Grounding**
   * **S1–S6:** Ensure every sentence is explicitly in the source, no speculation, focus on main points

**3. Prompt Template Design**

We split our prompt into **two** runtime templates under templates/:

* **system.md**  
  Holds **only** the H1–H3, P1, G1–G8, S1–S6 clauses—verbatim and without examples.
* **compliance\_eval.md**  
  A TIDD-style Chain-of-Thought evaluation template that:
  1. Lists the five compliance categories (adding “Illicit Behavior”).
  2. Defines severity levels and scoring weights.
  3. Includes a **Refusal Rule**: if Illicit Behavior = High → refuse and Grade=0.00/1.
  4. Outlines the exact **Output** format with six sections:
     + Summarization
     + Recommendations
     + Insights
     + Grade (0–1)
     + Critical Compliance Concern
     + Required Mitigation

This separation ensures the LLM sees only the guardrails as a system message, then the evaluation instructions + user input as the user message.

**4. API Implementation**

* **Configuration & Secrets**
  + Load .env safely, mask keys in logs, validate Azure endpoint, key, deployment.
* **Prompt Loading**
  + At startup read system.md and compliance\_eval.md into two strings.
* **/evaluate Endpoint**
  + Builds exactly two messages:
    1. **system** → system\_template
    2. **user** → filled-in compliance\_template (with [User input will be inserted here])
  + Calls Azure OpenAI with fixed max\_tokens, temperature, top\_p.
  + Parses the four required sections (plus two extra) via regex and returns a Pydantic model.

**5. Logging & Debugging**

Every request writes a JSON log to logs/<TIMESTAMP>.json containing:

jsonc

CopyEdit

{

"timestamp": "...",

"user\_input": "...",

"system\_prompt": "...",

"user\_prompt": "...",

"raw\_prompt": [ {role: "system",...}, {role: "user",...} ],

"api\_params": { model, max\_tokens, temperature, top\_p },

"raw\_output": "...",

"parsed\_output": { summarization, recommendations, insights, grade, critical\_compliance\_concern, required\_mitigation }

}

This lets you trace exactly what the LLM saw and produced.

**6. Test-Case Generation**

We crafted **domain-specific** SDLC & Networking prompts that:

* Cover all five categories
* Include both “simple” and “complex” examples
* Span five grade levels (1.00, 0.75, 0.50, 0.25, 0.00)
* Packaged into an Excel sheet with columns:
  + Category
  + Complexity
  + User Prompt
  + API Input (JSON blob)
  + Expected Grade

**7. Evaluation Framework**

We defined a structured directory and two scripts:

1. **evaluation/inputs/test\_cases.json**
   * Master list of test cases with **ground truth** fields for every section.
2. **evaluation/scripts/run\_tests.py**
   * Reads test\_cases.json, POSTs each to /evaluate, writes raw responses to  
     evaluation/outputs/raw\_responses/{TCID}.json.
3. **evaluation/scripts/evaluate\_results.py**
   * Loads ground truth & raw responses, computes per-case metrics:
     + **Grade error** = |gt\_grade − resp\_grade|
     + **Exact match** for summarization, critical concern, mitigation
   * Writes per-case eval to  
     evaluation/outputs/evaluated/{TCID}\_eval.json
   * Aggregates overall metrics into metrics/summary\_metrics.json, including:
     + Total tests, pass rate, average grade error