AI-Powered Regulatory Compliance Check & Gap Analysis for Nuclear Licensing

Repository Structure (High-Level)

```
/caelus_compliance_project
        /data
            /raw_pdfs             # Original regulatory PDFs
            /processed_text       # Extracted text from PDFs
            /fine_tuning_datasets # JSONL files for LLM fine-tuning (e.g., relations.jsonl, compliance_checks.jsonl)
        /models
            /embeddings           # Pre-trained Sentence Transformer model weights
            /fine_tuned_llm       # LoRA adapters for your LLM
        /src
            data_ingestion.py     # PDF parsing, text extraction, semantic unit creation
            llm_finetuning.py     # LoRA setup, training script
            knowledge_graph.py    # Relation extraction, graph building
            compliance_checker.py # Core compliance logic, LLM prompting
            report_generator.py   # Markdown output generation
        main.py                   # Orchestrates the pipeline
        demo_notebook.ipynb       # Jupyter Notebook for demonstration
        README.md                 # Project description, instructions, key findings
        requirements.txt          # Python dependencies
        .gitignore
```

Execution Plan: Detailed Steps (10 Working Days)

Phase 1: Setup & Data Foundation (Days 1-3)

Day 1: Project Setup, Research & Document Ingestion

Objective: Project environment setup, acquire target documents, basic PDF parsing.

Deliverables: Empty Git repo, requirements.txt, 5-7 selected nuclear PDFs, data/raw_pdfs populated, initial raw text files in data/processed_text.

Steps:

1. **Repository Initialization:**
   - Create `caelus_compliance_project` directory.
   - `git init`
   - Create `.gitignore` (add `__pycache__/`, `*.pt`, `*.bin`, `*.safetensors`, `venv/`, `env/`, `models/`, `data/fine_tuning_datasets/*.jsonl` if large).
2. **Environment Setup:**
   - Create a Python virtual environment (`python -m venv venv` or `conda create -n caelus_env python=3.9`).
   - Activate the environment.
   - Install core libraries: `pip install pypdf unstructured pdfminer.six torch transformers sentence-transformers faiss-cpu networkx scipy scikit-learn nltk spacy accelerate bitsandbytes peft trl`.
   - Create `requirements.txt` (`pip freeze > requirements.txt`).
   - Download `en_core_web_sm` or `en_core_web_lg` SpaCy model (`python -m spacy download en_core_web_lg`).
3. **Document Acquisition & Initial Research:**
   - Identify 5-7 specific, publicly available nuclear regulatory or technical documents (e.g., IAEA Safety Standards, NRC Regulatory Guides).
   - Place downloaded PDFs in `data/raw_pdfs/`.
   - Quickly read through them to understand their structure and content.
4. **src/data_ingestion.py - Raw Text Extraction:**
   - Implement a function to read PDFs and extract raw text using `pypdf` or `Unstructured`.
   - Save each document's raw text as a `.txt` file in `data/processed_text/`.

Day 2: Semantic Unit Segmentation & Embeddings

Objective: Break down raw text into manageable, meaningful "semantic units" (likely sentences or very short paragraphs) and generate their embeddings.

Deliverables: CSV or JSON file (`data/semantic_units.csv` or `.json`) containing (`unit_id`, `text`, `doc_id`, `section_title`, `page_number`, `original_order_in_doc`, `embedding_vector`).

Steps:

1. **src/data_ingestion.py - Semantic Unit Extraction:**

- Load raw text from `data/processed_text/`.
- Use `nltk.sent_tokenize` or `spaCy` for robust sentence segmentation.
- Metadata Extraction: Implement logic to extract `doc_id`, `section_title` (use regex or Unstructured's capabilities if available), `page_number`, and `original_order_in_doc` for each unit. This is vital for provenance.
- Store these units as a list of dictionaries.

2. **src/data_ingestion.py - Embedding Generation:**
   - Load a `SentenceTransformer` model (e.g., `sentence-transformers/BAAI/bge-large-en-v1.5`).
   - Generate embeddings for each semantic unit.
   - Add the `embedding_vector` to each unit's dictionary.
   - Save Semantic Units: Save the entire list of dictionaries (including embeddings) to a file (e.g., `data/semantic_units.pkl` or `data/semantic_units.json` if you convert embeddings to list/string).

Day 3: Design Specification Ingestion & Initial Structuring for Fine-tuning Data

Objective: Ingest a sample nuclear design specification document and begin preparing the dataset for LLM fine-tuning.

Deliverables: A simple, structured `design_spec.txt` or `design_spec.json`, and initial plan for fine-tuning dataset creation.

Steps:

1. **Acquire Design Spec:** Find a publicly available, small (e.g., 5-10 pages) nuclear design specification or technical description document (e.g., a specific component's design from an SMR vendor's public documentation, or an illustrative example from an NRC guide). Save as `data/design_spec.txt` or `data/design_spec.json`.

2. **src/data_ingestion.py - Design Spec Processing:**
   - Load `data/design_spec.txt`.
   - Break it into "design segments" (sentences or paragraphs describing a specific feature or function). Store with segment IDs.

3. **Fine-tuning Data Strategy:**
   - Review the parsed regulatory semantic units.
   - Primary Fine-tuning Focus for 2 Weeks: Relation Extraction from regulatory text. This directly helps build the graph and understand regulatory dependencies.
   - Secondary Fine-tuning Focus (if time allows): Compliance Judgment Explanation (for the prompt to decide compliance).
   - Plan for manual labeling of a small, high-quality dataset (e.g., 150-300 examples for each task).

Phase 2: LLM Fine-tuning & Knowledge Graph (Days 4-7)

Day 4-6: Dataset Generation for LLM Fine-tuning (Most Time-Consuming!)

Objective: Create high-quality, labeled datasets for fine-tuning your LLM, focusing on Relation Extraction.

Deliverables: `data/fine_tuning_datasets/relations.jsonl` (and `compliance_checks.jsonl` if pursuing).

Steps (Highly Manual/Semi-Automated):

1. **data/fine_tuning_datasets/relations.jsonl:**
   - From your `semantic_units.json` (regulatory clauses), manually select 150-300 representative clauses.
   - For each clause, identify key entities (e.g., "reactor coolant system," "safety classification") and their relationships.
   - Example format for `relations.jsonl` (one JSON object per line):

```
{
    "text": "The reactor coolant system shall be designed to withstand design basis accidents.",
    "entities": [
        {"span": "reactor coolant system", "type": "SYSTEM"},
        {"span": "design basis accidents", "type": "EVENT"}
    ],
    "relations": [
        {"head": "reactor coolant system", "type": "designed_to_withstand", "tail": "design basis accidents"}
    ]
}

```

   - Define your Relation Types clearly (e.g., `defines`, `requires_action`, `is_based_on`, `mitigates`, `applies_to`).
   - Tip: Use a simple labeling tool (like Prodigy's free trial, or even Google Sheets/Excel for structured export) or write a simple Python script to help. Consider using a larger LLM (like GPT-4) to propose initial labels for you to refine for faster dataset generation.
2. **(Optional but Recommended) data/fine_tuning_datasets/compliance_checks.jsonl:**
   - Pair `design_segment` with `regulatory_clause`.
   - Manually label `compliance_status` (`Compliant`, `Partially`, `Non-Compliant`) and provide a concise reasoning.
   - Example format:

```
{
    "regulatory_clause": "The pump's casing shall be
    capable of withstanding 500 psi.",
    "design_segment": "The pump casing is designed for 450
     psi.",
    "compliance_status": "Non-Compliant",
    "reasoning": "The design pressure of 450 psi is less
    than the required 500 psi."
}

```

Day 7: LLM Fine-tuning Setup & Execution

Objective: Fine-tune a pre-trained LLM using LoRA/PEFT on your created dataset.

Deliverables: Fine-tuned LoRA adapter weights saved in `models/fine_tuned_llm/`.

Steps:

1. **src/llm_finetuning.py:**
    - Load a base LLM (e.g., `mistralai/Mistral-7B-Instruct-v0.2` or `meta-llama/Llama-2-7b-chat-hf`) and its tokenizer.
    - Load the fine-tuning dataset(s) (`relations.jsonl`).
    - Configure BitsAndBytes for 4-bit quantization and PEFT (LoRA) for efficient fine-tuning.
    - Use TRL's SFTTrainer for easy instruction-tuned fine-tuning.
    - Define Prompts: Design prompts for the LLM during fine-tuning that align with how it will be used for inference (e.g., "Extract relations from the following text...").
    - Run the training. Monitor loss. Stop after 1-2 epochs or when validation loss plateaus (if you create a small validation split).
    - Save the LoRA adapters to `models/fine_tuned_llm/`.

Phase 3: Core Logic & Report Generation (Days 8-10)

Day 8: Knowledge Graph Construction from Fine-tuned LLM

Objective: Build a knowledge graph of regulatory requirements based on relations extracted by your fine-tuned LLM.

Deliverables: NetworkX graph object, potentially visualized (e.g., as a `.gml` or `.json` file).

Steps:

1. **src/knowledge_graph.py:**
    - Load your fine-tuned LLM and its tokenizer (load base model, then merge LoRA adapters).

- Iterate through all your `semantic_units` from the regulatory documents.
- For each unit, prompt the fine-tuned LLM to extract relations (using the prompt structure you fine-tuned it on).
- Use NetworkX to build a directed graph:
  - Nodes: Key concepts/entities extracted.
  - Edges: Relationships between them (e.g., "Clause A requires_action B").
  - Crucial: Attach the original `text_chunk` and `doc_id` as attributes to the nodes/edges they are derived from.
- Manual Graph Refinement (for demo quality): Given 5-7 documents, manually review the most critical regulatory requirements/concepts in your graph. Add 5-10 "golden" prerequisite/dependency links if the LLM missed them, ensuring the logical flow is perfect for the demo.
- Save the graph (e.g., `networkx.write_gml` or `networkx.write_json`).

Day 9: AI-Powered Compliance Checking Logic

Objective: Implement the core logic to compare design segments against regulatory clauses and generate compliance judgments.

Deliverables: Python functions for compliance checking.

Steps:

1. **src/compliance_checker.py:**
   - Load your fine-tuned LLM (or use a large commercial LLM if you didn't fine-tune for compliance judgment).
   - Load `semantic_units.json` (regulatory clauses) and `design_spec.json` (design segments).
   - Implement Similarity Search: For each `design_segment`, use FAISS to find the top-K (e.g., K=5) most semantically similar `regulatory_clauses`.
   - LLM-based Compliance Judgment: For each `design_segment` and its matched `regulatory_clause(s)`:
     - Prompt the LLM (using the prompt structure you fine-tuned for, or a robust general prompt):

```
Regulatory Requirement: [Reg Clause Text]
Design Specification: [Design Segment Text]
Is the design compliant? Provide a concise explanation
     and justification, citing specific parts of both
    texts if possible.

```

     - Output format:

```
{
    "compliance_status": "Compliant" | "Partially
    Compliant" | "Non-Compliant",
    "reasoning": "...",
    "regulatory_citation": "...",
    "design_citation": "..."
}

```

  - Store these compliance judgments in a structured list.

Day 10: Report Generation & Final Output

Objective: Generate the final, professional Markdown compliance report.
Deliverables: `compliance_report.md` in the root directory.
Steps:

1. **src/report_generator.py:**
   - Function to assemble the Markdown report.
   - Structure:
     - Project Title, Introduction.
     - Overview of the Design Specification.
     - Section for each `design_segment` from the input.
     - For each `design_segment`:
       * Display the segment text.
       * State the `compliance_status`.
       * Present the reasoning generated by the LLM.
       * Provide full `regulatory_citation(s)` for all matched clauses.
       * (Optional but great) Include the text of the matched regulatory clauses for context.
     - Summary of overall compliance status (e.g., "X% Compliant, Y% Partially Compliant, Z% Non-Compliant").
     - A section highlighting all identified Non-Compliant and Partially Compliant items for quick review.
     - Conclusion.
   - Write the content to `compliance_report.md`.
2. **main.py:** Create a simple script that orchestrates the entire pipeline from data ingestion to report generation.
3. **demo_notebook.ipynb:** Create a clean, step-by-step Jupyter Notebook that:
   - Explains the project goal.
   - Shows how to load and process regulatory docs and design spec.

- Demonstrates the fine-tuning process (or just loads the fine-tuned model).
- Executes the compliance checking for a few illustrative examples.
- Displays snippets of the generated report.
- Clearly showcases the value proposition.

4. **README.md:** Write a powerful README that covers the "Impress Them" factors outlined earlier. Emphasize problem addressed, your innovative AI approach (especially LLM fine-tuning), the specific nuclear domain, and the demonstrable benefits (automation, accuracy, time-saving). Be transparent about the PoC nature and future work.

Important Reminders for Solo Execution

- **Stay Focused:** Do not get sidetracked by perfecting one module beyond what's needed for the PoC.
- **Timeboxing:** Stick to the daily breakdown as much as possible. If a task runs over, identify the minimum viable output for that task and move on.
- **Test as You Go:** Implement small tests for each module (e.g., "does PDF parsing extract text correctly?", "does LLM fine-tuning run without errors?").
- **Leverage Existing Code:** Re-use snippets from online tutorials or your previous projects where possible.

This plan is aggressive but achievable for a highly motivated and skilled individual in 10 working days. The key is strict scope management and a focus on the core "impressive" demonstrations. Good luck!