Sustainable AI – Energy-Aware Prompt & Context Engineering

Technical Design & Architecture Report

Group 2

Abdullahi Abdirizak Mohamed - 9082466

Albright Maduka Ifechukwude - 9053136

Jose George - 9082825

Kamamo Lesley Wanjiku - 8984971

1. Introduction

Large Language Models (LLMs) can be expensive and energy-intensive to run, especially when prompts are long, repetitive, or poorly structured. The Sustainable AI project is a proof-of-concept system that estimates the energy impact of an LLM configuration and prompt, suggests a more concise or energy-efficient alternative, and flags anomalously high-energy usage patterns through an interactive UI.

2. High-Level Overview

- Prompt Processing & Optimization – parses structured prompts (Role, Context, Expectations), computes simple linguistic features, and generates a shorter optimized prompt.
- Energy Estimation Engine – estimates approximate energy consumption (kWh) based on model configuration and prompt length, using either a trained regression model or a heuristic fallback.
- Anomaly Detection Module – uses an Isolation Forest trained on synthetic data to flag configurations that deviate from normal energy/compute patterns.
- Visualization & UI Layer – Streamlit app that connects the components and visualises metrics, distributions, and anomaly results.
- Data & Model Artifacts – synthetic datasets and persisted sklearn pipelines used for training and inference.

3. Codebase Structure

The main components are organised as follows:

• data/: Synthetic CSVs for energy modelling and anomaly detection.
• model/anomaly_detector/: Serialized IsolationForest pipeline (iso_pipeline.pkl).

• src/app.py: Streamlit UI and high-level orchestration.
• src/energy_model.py: Wrapper for the energy regression or heuristic model.
• src/prompt_optimizer.py: Rule-based prompt simplification and complexity metrics.
• src/anomaly_detector.py: Wrapper around the IsolationForest anomaly detector.
• src/visualization.py: Reusable matplotlib/seaborn visualisations.
• src/models.py: Dataclasses for structured prompt fields.
• notebooks/: Jupyter notebooks for model training and experimentation.

4. Data & Feature Engineering

The project uses synthetic data to approximate LLM energy usage. Each row of the dataset represents a configuration with fields such as number of layers, training hours, FLOPs per hour, prompt length, and synthetic energy consumption (kWh).

For anomaly detection, additional features are engineered:
• flops_per_layer = flops_per_hour / num_layers
• energy_per_hour = energy_kwh / training_hours
• energy_per_token = energy_kwh / prompt_tokens

These, along with the original configuration features, form the feature vector for the Isolation Forest model.

5. Prompt Processing & Optimization

Prompts are represented as three sections—Role, Context, Expectations—via a PromptData dataclass. The optimizer takes a combined structured prompt and:

- Parses it back into Role, Context, Expectations if the labels are present.
- Simplifies each section by normalising whitespace, splitting into sentences, keeping only the first 2–3 key sentences, and removing common filler phrases.
- Reconstructs a structured prompt with Role, Context, Expectations labels to keep the format consistent for the UI.
- Computes simple complexity metrics (token count, average sentence length, stopword ratio, number of sentences).
- Approximates semantic similarity between original and simplified prompts using Jaccard overlap of non-stop word tokens.

6. Energy Estimation Engine

The energy engine can use either a trained regression model or a heuristic fallback. In the current proof-of-concept, a heuristic is favoured for its stability and interpretability. The fallback model computes energy as a combination of a training term and a prompt term:

• Training energy is proportional to num_layers × training_hours × (flops_per_hour / 1e20).
• Prompt energy is proportional to a logarithmic function of token count.
• The final value is clipped to a reasonable kWh range (e.g., 0.1–30) for display.

This ensures that increasing layers, training time, FLOPs, or prompt length increases the estimated energy, and that shorter optimized prompts visibly reduce the estimated energy compared to the baseline.

7. Anomaly Detection Module

The anomaly detector is trained in a dedicated notebook (`anomaly_detection.ipynb`). It fits an Isolation Forest inside a Pipeline with a StandardScaler over the engineered feature matrix X. The contamination parameter controls the fraction of configurations treated as anomalies (e.g., 5%).

At runtime, `src/anomaly_detector.py` exposes a `flag_usage_anomaly(record)` function that:

- Builds a single-row DataFrame with the required features and engineered values.
- Uses the fitted pipeline's decision_function/score_samples to compute an anomaly score.
- Calls predict() to map the record to normal (1) or anomalous (-1).
- Returns a dictionary with is_anomaly (bool), score (float), and a short textual explanation.

8. Visualisation Layer

The visualisation module builds reusable charts:

- Before vs After feature comparison bar chart (tokens, average sentence length, stopword ratio, sections).
- Energy usage histogram across previous runs, with a vertical line marking the current prompt's energy.
- Token breakdown bar chart by prompt section (Role, Context, Expectations).
- Anomaly score bar chart indicating whether the current configuration is normal or anomalous.

In addition, the Streamlit app renders a small 'Energy Prediction vs Actual (Mocked)' curve based on a toy regression example, with a marker showing where the current configuration sits on the curve.

9. Streamlit UI & User Flow

The front-end is implemented in `src/app.py` using Streamlit. The layout has a hero section on the left (branding and description) and an interactive prompt and parameter card on the right. The main user flow is:

- User enters Role, Context, Expectations and sets model parameters (# layers, training hours, FLOPs/hr).
- On Submit, the app constructs the full prompt, estimates token count, and calls the energy engine for a baseline estimate.
- The prompt optimizer generates an alternative version and the app recomputes energy for the optimized prompt.
- The anomaly detector evaluates the configuration and returns a score and flag.
- The results section shows baseline vs alternative energy, original vs optimized prompts, complexity metrics, history histogram, token breakdown, and anomaly visuals.
- On Improve, the optimized prompt is parsed back into Role, Context, Expectations and written into the text areas so the user can iterate.

10. Limitations & Future Work

The current implementation is intentionally lightweight and pedagogical. Energy estimates are heuristic and based on synthetic data rather than real sensor readings. For prediction tasks, we have used only regression models so far; however, we intend to incorporate neural networks in future versions to achieve more accurate and nuanced predictions. Prompt similarity and complexity metrics are shallow compared to embedding-based methods. In a production system, future work could include:

- Integrating real GPU/TPU energy telemetry and CO_2 emission factors.
- Replacing heuristic energy models with calibrated regression or physics-informed models.
- Utilizing neural networks to enhance the precision of energy and anomaly predictions.
- Using sentence embeddings and cosine similarity for robust semantic comparisons.
- Collecting real workload logs to training a more realistic anomaly detector.
- Extending the UI to support batch analysis of multiple prompts and configurations.

System Architecture Overview

The Sustainable AI system follows a modular, layered architecture. At a high level, it consists of:

- **User Interface Layer (Streamlit app)**

    - Collects user inputs: Role, Context, Expectations, number of layers, training time, FLOPs/hour.
    - Displays energy estimates, optimized prompts, anomaly flags, and visualizations.
- **Application / Orchestration Layer (app.py)**
    - Coordinates calls to the prompt optimizer, energy engine, anomaly detector, and visualization utilities.
    - Implements the main user flow for **Submit** and **Improve** actions.
    - Logs per-run energy values to a CSV file for historical analysis.
- **Core Services Layer**
    - **Prompt Processing & Optimization (prompt_optimizer.py)**
        - Parses structured prompts (Role, Context, Expectations).
        - Simplifies each section (removing filler, reducing repetition).
        - Computes basic complexity metrics and an approximate semantic similarity score.
    - **Energy Estimation Engine (energy_model.py + heuristic in app.py)**
        - Uses model configuration and prompt length (tokens) to estimate energy in kWh.
        - Falls back to a heuristic formula if a trained regression model is not available.
    - **Anomaly Detection Module (anomaly_detector.py)**
        - Loads a pre-trained Isolation Forest pipeline.
        - Builds a feature vector from a single configuration.
        - Returns an anomaly score, a Boolean flag, and a short explanation.
- **Visualization Layer (visualization.py)**
    - Generates reusable plots:
        - Before vs After feature comparison (tokens, avg sentence length, stop word ratio, sections).
        - Historical energy usage histogram with a marker for the current run.
        - Token breakdown per prompt section (Role, Context, Expectations).
        - Anomaly score bar indicating normal vs anomalous usage.

- **Data & Model Artifact Layer**
    - Stores synthetic datasets and the trained anomaly detection pipeline.
    - Persists historical energy logs generated by the UI.

Component-Level Architecture

1. User (Browser)

- Enters Role, Context, Expectations.
- Chooses model parameters: number of layers, training hours, FLOPs/hour.
- Triggers analysis via **Submit** and refinement via **Improve**.

2. Streamlit Frontend (src/app.py)

- Build the full prompt from the three sections.
- Estimates prompt token count using a simple word-based heuristic.
- Calls:
    - optimize_prompt(...) from prompt_optimizer.py
    - safe_predict_energy(...) which wraps the energy model or heuristic
    - flag_usage_anomaly(...) from anomaly_detector.py
- Logs energy_kwh values to energy_logs.csv.
- Invokes plotting functions from visualization.py and renders them in the UI.

3. Prompt Optimizer (src/prompt_optimizer.py)

- Input: structured or combined prompt text.
- Steps:
    - Parse Role / Context / Expectations if labels are present.
    - Normalize whitespace and split each section into sentences.
    - Keep only the most important 2–3 sentences per section.
    - Remove common filler phrases (e.g., “please”, “kindly”, “I would like you to…”).
    - Reconstruct a shorter, still structured prompt with the same labels.

    - Compute complexity metrics before and after optimization and a similarity score.

- Output:
    - Original prompt, simplified prompt, complexity_before, complexity after, semantic similarity.

4. Energy Estimation Engine (src/energy_model.py + heuristic in app.py)

- Input features:
    - num_layers
    - training_hours
    - flops_per_hour
    - prompt_tokens
- Logic:
    - Normalize configuration against a "typical" setup.
    - Compute a **training term** proportional to layers × hours × FLOPs.
    - Compute a **prompt term** that grows with the logarithm of token count.
    - Combine and clip to a realistic kWh range (e.g., 0.1–30 kWh).
- Output:
    - A single scalar energy estimate is used for baseline and optimized prompts.

5. Anomaly Detection Module (src/anomaly_detector.py)

- Training (in the notebook):
    - Uses synthetic data with features such as:
        - num_layers, training_hours, flops_per_hour, prompt_tokens, energy_kwh
        - engineered features: flops_per_layer, energy_per_hour, energy_per_token
    - Fit an IsolationForest inside a Pipeline with StandardScaler.
    - Saves the fitted pipeline to: model/anomaly_detector/iso_pipeline.pkl.
- Runtime:
    - flag_usage_anomaly(record):
        - Builds a single-row Data Frame with all required features.

- Applies the pipeline's decision function or score samples to compute a score.
- Uses predict () to label the record as normal (1) or anomalous (-1).
- Returns a dictionary with:
    - is anomaly (Boolean),
    - score (float),
    - explanation (short text).