Final Project Description

For the final project, students are required to **design** and **implement** an AI-driven system with a strong emphasis on cloud-based architectures and sustainable deployment practices. Each project must be implemented and demonstrate a coherent, modular, and well-documented architecture.

A typical project should consist of:

- **Architecture Decision Records (ADRs)** to justify critical design choices.
- **UML class and sequence diagrams** for the main use cases (architecturally significant use-cases).
- **CI/CD pipeline** to automate testing, building, and deployment.
- **Observability and monitoring dashboards** that capture system health, performance, and AI behavior.
- **Sustainability considerations in deployment**, including trade-offs between scalability, energy efficiency, and cost.
- **Explicit carbon metrics** (e.g., CO_2e per request, job) using available Python libraries.
- **Carbon-aware behavior** driven by a simulated carbon-intensity signal.
- **Automatic redeployment or routing** of services/models for efficiency under changing simulated carbon conditions, while maintaining defined **SLOs (Service Level Objectives)**.

Project Choice and Presentation

The proposed project ideas can be used as inspiration or as starting points for development. Students are also **encouraged to suggest** and design systems that are relevant to their own professional context or aligned with ongoing personal or workplace projects.

Assessment will consist of two main components:

1. **The project itself**, evaluated on technical quality, architectural design, sustainability considerations, and operational practices.
2. **The presentation**, in which students must demonstrate not only the system implementation but also the application of theoretical principles covered in the course lectures (e.g., architectural patterns, sustainability trade-offs, or cloud-native design strategies).

Example projects

Note:

1) The students have full freedom on implementation details and choices. Usage of modern coding principles (code scaffolding, modeling, generation and generative AI) is allowed. However, the student needs to demonstrate COMPLETE understanding of the entire code base, and capable of following and explaining any code block.

2) In this course, we don not train models, optimize LLMs, adjust hyper-parameter tuning, etc. This course focuses on engineering solutions using modern AI principles. That said, we architect solutions, we use "lego bricks" and develop prototypes with sustainability concerns. Component based software engineering principles are highly encouraged, re-usability and focus on interfacing and integration.

- **NLP Helpdesk Triage Bot:** Implement a Python service that classifies incoming tickets and suggests responses using a lightweight transformer or classical ML baseline.
  - Design a modular cloud architecture (API gateway, inference service, feature store) and capture decisions in ADRs with UML diagrams for ticket → classify → response.
  - Establish CI/CD that runs unit tests, model checks, and builds containers; support two deployment profiles: *fast/high-power* and *eco/low-power*.
  - Measure inference carbon footprint, and make routing carbon-aware: under high carbon intensity, route to distilled or quantized models; under low intensity, allow full-precision models.
  - Provide a simulation harness that toggles carbon intensity and ticket volume to trigger automated rollout between profiles, with dashboards for accuracy, latency, cost, and gCO_2e/ticket.
- **Real-Time Anomaly Detection for IoT Streams:** Build a Python microservice that ingests sensor streams (e.g. simulate with Kafka or MQTT) and flags anomalies using an online model (e.g., Isolation Forest).
  - Architect separate services for ingestion, inference, feature extraction, and a REST API; document ADRs and provide UML class + sequence diagrams.

    - Implement CI/CD to run tests, containerize, and deploy to two environments: a GPU-enabled region and a CPU-only, low-carbon region.
    - Integrate a carbon-measurement library (e.g., CodeCarbon) and a carbon-intensity signal (mock or rules file) to make the service carbon-aware.
    - Simulate *low/high grid* intensity and automatically shift traffic or redeploy to the more efficient environment when a carbon/SLO threshold is crossed; expose metrics and alerts (latency, throughput, anomaly rate, CO_2e/inference).
- **Vision-Based Quality Inspection (Edge + Cloud):** Create an edge inference service (simulated with local containers) that scores product images on flagged defects.
    - Architect edge upload, cloud storage, model registry, staged deployment. Include ADRs and UML for model promotion and rollback.
    - CI/CD must support artifact versioning and staged gradual rollouts (canary approach) to edge nodes (to 10%, then 50%, then 100%).
    - Track energy/carbon for edge inference; when simulated carbon intensity spikes in the cloud region, defer or move it to a greener region/time window.
    - The simulation flips between *low and high intensity* and forces the controller to reschedule jobs and switch edge models (e.g., from ResNet-50 to MobileNet) while keeping SLOs visible in dashboards (throughput, false-rejects, gCO_2e/frame).
- **Time-Series Forecasting for Demand Planning:** Build a forecasting service (e.g., Prophet/ARIMA/lightweight RNN) behind a REST API that returns next-day demand for multiple stock keeping units (SKU).
    - Separate components for data ingest, feature engineering and inference; provide ADRs and UML sequence diagrams for prediction workflows.
    - Implement CI/CD with reproducible data/versioned models and infrastructure-as-code; support two deployment targets: serverless (bursty, scale-to-zero) and long-running containers (warm, higher perf).
    - Instrument carbon measurement and add a carbon-aware scheduler: when simulated grid intensity is high, prioritize serverless batch inference;
    - The simulation varies carbon intensity and workload size to trigger automatic environment switching and model quantization, with observability covering MAPE/MAE, cold-start rate, cost, and gCO_2e/forecast.
- **Recommendation API for Course Content:** Implement a Python recommender (matrix factorization or implicit feedback) served via an API that returns top-N items per user.
    - Architect services for data ETL, candidate generation, ranking, and a model registry; capture ADRs and UML diagrams.
    - CI/CD pipelines run data/unit tests, build containers, and deploy to two regions with different instance classes; add feature flags for model precision (FP32 vs. int8) and caching strategy.
    - Use a carbon-tracking library and a simulated carbon-intensity feed to make the router carbon-aware: under high intensity, switch to cached recommendations + int8 inference; under low intensity, enable full re-ranking.

  - The simulation alternates *low/high intensity* to force automated redeploy/canary and traffic shifting, while dashboards report hit-rate/NDCG, p95 latency, cache hit ratio, cost, and gCO_2e/request.

- **AI-Powered Incident Response Classifier:** Develop a system that automatically classifies IT support incidents (network, software, hardware, security) using NLP models.
  - The architecture must include ticket ingestion (API or queue), model inference service, and a dashboard for metrics.
  - Students must implement CI/CD for redeployment, and add monitoring for classification accuracy, latency, and throughput.
  - Carbon measurement should capture emissions per inference. Under simulated high-carbon conditions, the system should automatically route requests to a lighter model (e.g., distilled transformer) while still maintaining defined SLOs.
- **AI-Enhanced Predictive Maintenance System:** Build a predictive maintenance solution for industrial machines using simulated sensor data (vibrations, temperature, RPM).
  - The architecture should include data ingestion, feature extraction, anomaly detection, and predictive modeling.
  - CI/CD must ensure that models and pipelines are versioned and redeployed automatically when updates occur.
  - Students should measure the carbon footprint of inference jobs and make the pipeline carbon-aware.
  - When simulated carbon intensity is high, inference should be deferred or shifted to greener regions, or it should downscale to a lightweight fallback model.
- **Personalized News Recommendation Engine:** Create a recommender system that suggests articles to users based on their reading history and preferences.
  - The architecture must support data ETL, candidate retrieval, ranking, and logging feedback loops.
  - Students must provide ADRs and UML diagrams, implement CI/CD for continuous deployment of ranking models, and add observability for metrics like NDCG and latency.
  - Carbon measurement must track gCO_2e per recommendation. Simulated carbon-intensity signals should trigger automatic model switching (full vs. quantized model) or cache-based recommendation serving.
- **Fraud Detection Microservice for Transactions:** Implement a fraud detection engine that scores financial transactions in real time using an ML classification model.
  - The architecture should include a transaction stream simulator, a scoring service, and a dashboard showing fraud probability and system health.
  - CI/CD must support testing, and safe rollout of updated models with canary deployments. Students must integrate carbon accounting to track emissions per 1000 transactions.
  - When carbon intensity is simulated to be high, the system should switch to lower-cost inference instances, while ensuring latency SLOs are met.
- **AI-Based Traffic Flow Optimizer:** Design a system that predicts traffic congestion and suggests optimal routing using simulated traffic sensor and GPS data.

- The architecture should include ingestion of traffic feeds, a predictive model, and an API for serving routing recommendations.
- CI/CD pipelines must support automated deployment of updated models and infrastructure-as-code provisioning.
- Carbon footprint metrics must be reported for inference and data ingestion. When carbon intensity is simulated as high, the system should reroute inference workloads to greener regions or deploy lightweight routing models, while still delivering recommendations within latency SLOs.