Specialisation Project (VT1)
HS2024

Platform for Investment Analysis

Linear Programming Optimization Model for Integrated Energy Systems in Python

Submitted by
Rui Vieira
Institute of Product Development and Production Technologies (IPP)

Supervisor
Dr. Andrea Giovanni Beccuti
IEFE Model Based Process Optimisation

Study Program
Business Engineering, MSc in Engineering

Zurich University of Applied Sciences
School of Engineering

February 15, 2025

Imprint

Project: Specialisation Project (VT1)
Title: Platform for Investment Analysis
Author: Rui Vieira
Date: February 15, 2025
Keywords: linear programming, quantitative modeling, python, strategic planning, optimization, asset valuation, power-flow, platform

Study program:
Business Engineering, MSc in Engineering
ZHAW School of Engineering

Supervisor:
Dr. Andrea Giovanni Beccuti
IEFE Model Based Process Optimisation
Email: giovanni.beccuti@zhaw.ch

Abstract

This work presents an integrated investment framework for energy systems, focusing on optimal technology selection and placement of electrical generation, conversion, and storage assets. The core engine combines DC Optimal Power Flow (DC-OPF) simulations with linear programming (using the PuLP solver) to evaluate both technical feasibility and economic viability across a multi-scenario analysis. A 9-bus test network provides the backdrop for a reduced ten distinct cases, each featuring a unique mix of conventional (nuclear, gas) and renewable (solar, wind) power plants, supplemented by battery storage of varying capacities.

To balance computational efficiency with seasonal realism, the annual horizon is divided into three representative weeks (summer, winter, and spring/autumn), whose costs and operations are subsequently scaled to form a full-year analysis. This approach reveals significant seasonal differences in storage utilization even enabling clean assets to compensate for the cost of conventional generation.

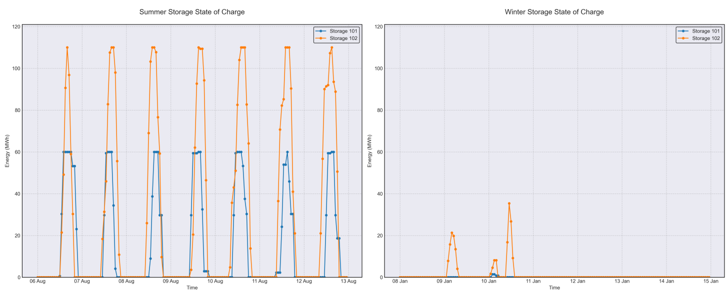


Figure 1: Summer/Winter battery SoC profile comparison – Scenario 5

In scenarios featuring abundant solar generation, the SoC frequently reaches its upper limits, highlighting the potential for upsizing or more flexible operational strategies—such as battery leasing or modular additions—to capture peak renewable output.

An economic sensitivity analysis underscores the strong influence of high-cost resources during extreme load conditions, causing a disproportionate rise in total costs when reliance on expensive generation escalates. Meanwhile, scenarios with nuclear-dominated baseload exhibit lower operational cost volatility but may still benefit from targeted storage deployment to manage residual demand swings. AI-assisted reporting consolidates these findings by identifying cost drivers, optimal technology mixes, and operational bottlenecks across all scenarios. Notably, Scenario7's balanced blend of nuclear, solar, and wind with moderate battery support emerges as the most cost-effective configuration, while Scenario4, featuring gas-fired generation and multiple storage units, proves the least favorable in terms of net present value (NPV).

Overall, the proposed framework bridges technical dispatch simulation and investment analysis, guiding stakeholders in designing resilient, economically viable energy systems. Future enhancements include broader maintenance modeling, real-time price integration for advanced arbitrage strategies, and further prompt-engineering improvements to refine AI-driven reporting and decision support.

Keywords: linear programming, quantitative modeling, python, strategic planning, optimization, asset valuation, power-flow, platform

Contents

1 Introduction

1.1 Project Context

Energy systems are increasingly critical in modern society due to rising electricity demand and the global push toward cleaner energy sources. Power flow studies, which evaluate how electricity moves through transmission networks, identify optimal generation mixes, ensure technical feasibility and able to optimize the location of assets.

In addition to analyzing power flows, it incorporates financial metrics and the analysis of various generation assets, including renewables and storage ones. It underscores the ongoing importance of power flow problems in both daily grid operations and strategic energy planning.

Undertaken as a semester project (so-called specialisation project) for the MSE program, it builds mainly on methodologies from life cycle management and optimization [5] courseworks. The project offers a reproducible demonstration of how technical and economic feasibility can be linked.

1.2 Objectives

The project sets three main objectives:

1. Develop a DC Optimal Power Flow (DCOPF) solver capable of handling multi-scenario analyses with variable generation and storage configurations.
2. Implement an investment analysis approach
3. Demonstrate a comparative method for evaluating different resource mixes under diverse load profiles and cost assumptions.

The main challenge here is to compare multiple generation and storage configurations within an existing electrical grid. On the technical side, a *DC power flow* is formulated and solved via linear programming, yielding hourly dispatch decisions under network constraints. While various objective functions could be considered—such as emissions reduction—this implementation focuses solely on cost minimization, subject to physical network constraints.

For the economic dimension, we opted to implement an *investment analysis* framework—focusing on Net Present Value (NPV), annuity, and a load-based sensitivity analysis to gauge long-term costs across different scenarios. This integrated perspective provides a structured way to identify cost-effective integration of renewable and storage assets.

2 Theoretical Background

2.1 Linear Programming in Energy Systems

At its core, our problem focuses on meeting electricity demand through optimal generation dispatch: determining how much power each generator should produce to satisfy consumer demand while minimizing costs and respecting system constraints. This fundamental power systems challenge can be effectively modeled using Linear Programming (LP).

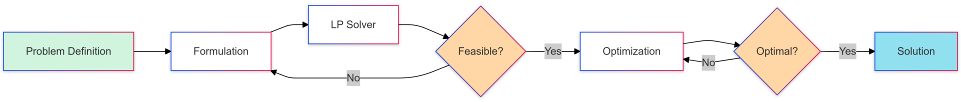


Figure 2: LP optimization flowchart showing key steps from problem formulation to optimal solution.

Linear Programming enables modeling of key power system relationships - such as power balance (matching generation to demand), transmission limits, and generation constraints - as linear equations and inequalities. The basic structure of our optimization problem is:

- **Objective**: Minimize total generation cost
- **Primary Decision**: How much power to generate at each plant
- **Key Constraint**: Total generation must meet demand at all times
- **System Constraints**: Respect network and equipment limitations

This can be expressed mathematically as:

$$\min_{\mathbf{x}} \mathbf{c}^\top \mathbf{x} \quad \text{subject to} \quad A\mathbf{x} \leq \mathbf{b} \tag{1}$$

2.2 DC Optimal Power Flow

The DC Optimal Power Flow (DC-OPF) extends the basic generation dispatch problem by incorporating network constraints. It answers the question: "How should we distribute power generation across the network to meet demand at minimum cost while respecting transmission line limits?" The DC-OPF achieves this by:

- Modeling power flow through transmission lines
- Ensuring power balance at each network node
- Respecting both generation and transmission limits

The "DC" prefix indicates a linearized approximation of the full AC power flow equations, making the problem solvable using LP techniques [9]. This approximation is particularly effective for high-voltage transmission planning [1].

2.2.1 Key Assumptions

The DC approximation makes four key simplifications:

- Voltage magnitudes are fixed at 1.0 per unit

- Line resistances are negligible ($R \ll X$)
- Voltage angle differences are small
- Reactive power (power that oscillates between source and load without doing useful work) is ignored

These assumptions yield a simple relationship between power flow (P_{ij}) and voltage angles (θ):

$$P_{ij} = B_{ij}(\theta_i - \theta_j) \tag{2}$$

2.2.2 Mathematical Formulation

The DC-OPF problem minimizes generation costs subject to network constraints. Each equation represents a physical aspect of power system operation:

1. Power Balance - The fundamental law of power systems

- At each bus i, power in equals power out
- Generation minus demand equals net power flow to neighboring buses
- Determined by line susceptances and voltage angles

$$\sum_{g \in \mathcal{G}_i} P_{g,t} - D_{i,t} = \sum_{j \in \mathcal{N}_i} B_{ij}(\theta_{i,t} - \theta_{j,t}) \quad \forall i \in \mathcal{N}, t \in \mathcal{T} \tag{3}$$

2. Cost Minimization - Economic objective

- Find optimal generation dispatch that minimizes total system cost
- Each generator has an associated marginal cost function (cost per unit of production)

$$\min_{\mathbf{P_g}, \boldsymbol{\theta}} \sum_{g \in \mathcal{G}} \sum_{t \in \mathcal{T}} c_g P_{g,t} \tag{4}$$

3. Line Capacity - Network limitations

- Power flow must stay within thermal limits of transmission lines
- Bi-directional constraint (forward and reverse flow limits)

$$-P_{ij}^{\max} \leq B_{ij}(\theta_{i,t} - \theta_{j,t}) \leq P_{ij}^{\max} \quad \forall (i,j) \in \mathcal{L}, t \in \mathcal{T} \tag{5}$$

4. Generator Limits - Physical constraints

- Each generator is limited by minimum and maximum output
- Generators may have time-varying limits

$$P_g^{\min} \leq P_{g,t} \leq P_g^{\max} \quad \forall g \in \mathcal{G}, t \in \mathcal{T} \tag{6}$$

5. Reference Angle - System reference

- One bus sets the reference for voltage angles
- Typically chosen as the largest generator

$$\theta_{\text{slack},t} = 0 \quad \forall t \in \mathcal{T} \tag{7}$$

2.2.3 Storage System Constraints

The model includes battery storage systems with the following constraints:

1. Energy Balance - Storage state evolution

- Tracks energy level over time
- Accounts for charging and discharging efficiencies

$$E_{s,t+1} = E_{s,t} + \eta_c P_{c,s,t} - \frac{P_{d,s,t}}{\eta_d} \quad \forall s \in \mathcal{S}, t \in \mathcal{T} \tag{8}$$

2. Power Limits - Operational boundaries

- Maximum charging and discharging rates
- Cannot charge and discharge simultaneously

$$0 \leq P_{c,s,t} \leq P_{c,s}^{\max} \quad \forall s \in \mathcal{S}, t \in \mathcal{T} \tag{9}$$
$$0 \leq P_{d,s,t} \leq P_{d,s}^{\max} \quad \forall s \in \mathcal{S}, t \in \mathcal{T} \tag{10}$$

3. Energy Capacity - Storage limits

- Maximum and minimum state of charge
- Often includes end-state condition

$$E_s^{\min} \leq E_{s,t} \leq E_s^{\max} \quad \forall s \in \mathcal{S}, t \in \mathcal{T} \tag{11}$$
$$E_{s,T} = E_{s,0} \quad \forall s \in \mathcal{S} \tag{12}$$

Where:

- $E_{s,t}$: Energy stored in battery s at time t
- $P_{c,s,t}, P_{d,s,t}$: Charging and discharging power
- η_c, η_d: Charging and discharging efficiencies

3 Methodology and Implementation

3.1 System Architecture

3.1.1 Data structure and Input Flow

The architecture relies on a set of CSV files that define (1) the physical network, (2) the time-series input data, and (3) the scenario configurations for analysis. All of these files reside in the `data/working` directory and are ultimately passed to the `main.py` script, which processes each scenario in turn.

1. **Physical Network Files**: `branch.csv` and `bus.csv`
   Define the perimeter and physical layout of the power grid in a format inspired by MATPOWER [6]. For example, `branch.csv` provides line impedances and flow limits, while `bus.csv` specifies bus voltage information, types, and identifiers.
2. **Time-Series Input Data**: `master_gen.csv` and `master_load.csv`
   One holds the asset-level generation profiles (nuclear, wind, solar, etc.), including capacity limits, per-unit costs, and operational constraints.

   The other one, contains bus-level load profiles, mapped to specific seasonal segments (winter, summer, spring/autumn). These files reflect the availability or demand data relevant to a particular study-case.
3. **Scenario Configurations**: `scenarios_parameters.csv`
   Centralizes the definitions of each scenario: which generators or storage units are placed at which buses, along with any load scaling factors (e.g., ±20% demand). This dataset dictates how the solver will allocate and dispatch resources in different configurations.

The main driver script, `multi_scenario.py`, loads and combines these CSVs to set up each scenario's DCOPF problem. By separating network topologies, time-series inputs, and scenario definitions, the framework allows new assets, bus layouts, or experimental conditions to be tested without significant changes to the core codebase.

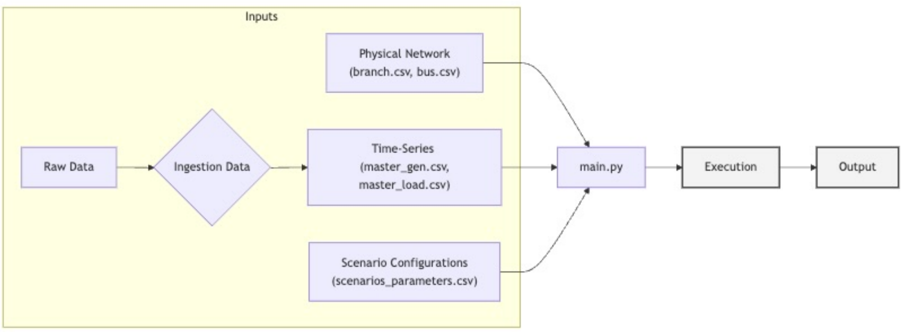


Figure 3: Overview of the Data Flow into the Optimization Process

3.1.2 Execution

The `main.py` script serves as the central orchestrator of the optimization workflow. Its key responsibilities include:

- **Scenario Management**
  Loads scenario definitions from `scenarios_parameters.csv`, which specify generator placements, storage configurations, and load scaling factors for each test case.

- **Data Integration**
  Combines network topology data (`bus.csv`, `branch.csv`) with time-series inputs (`master_gen.csv`, `master_load.csv`) to construct complete optimization problems.

- **Sensitivity Analysis**
  For each base scenario, optionally generates variants with modified load factors (e.g., ±20%) to test system robustness under different demand conditions.

- **Results Collection**
  Aggregates solver outputs, calculates key metrics (e.g., capacity factors, annual costs), and stores results in standardized formats for further analysis.

- **Investment Analysis**
  Interfaces with `create_master_invest.py` to compute financial metrics like NPV and annualized costs across different time horizons.

They can be vizualized in the following flowchart:

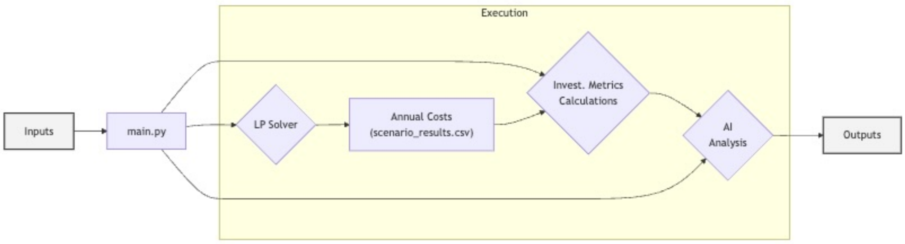


Figure 4: Execution and interaction of the main.py script with the other modules

The `main.py` script also coordinates with auxiliary modules for visualization (`summary_plots.py`), and documentation updates (`update_readme.py`), ensuring up-to-date visualizations and online-documentation.

3.1.3 Output Flow

After `main.py` coordinates the execution of all scenarios through `dcopf.py`, `create_master_invest.py`, and `summary_plots.py`, it generates three key outputs:

- **Global Summary Report**
  A comprehensive `summary.md` file that ranks scenarios by annuity value, provides AI-generated insights on overall trends, and includes comparative visualizations across scenarios.

- **Individual Scenario Reports**
  For each scenario (e.g., `scenario_1_analysis.md`), it generates a detailed report with: dispatch plots and generation mix charts, financial metrics breakdown, and AI-generated commentary on specific operational patterns.

- **Consolidated Results**
  A `scenario_results.csv` within `data/results` containing raw operational data (generator dispatch, line flows), investment metrics (NPV, annual costs, annuities), and sensitivity analysis results (if enabled).

It illustrates the principal output files after interaction with the main.py script and the other modules.

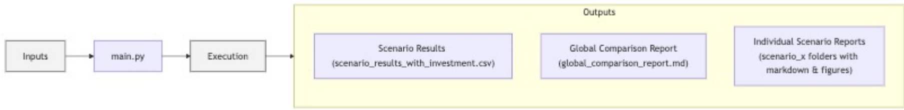


Figure 5: Output results and reports

This multi-step workflow ensures all relevant data—raw operational outputs, investment metrics, and optional AI insights—remain easily accessible for post-processing or stakeholder review. As a result, users can quickly compare scenarios under different configurations, load sensitivities, or asset placements without altering the core solver routines.

3.2 Core Components

We detail the four principal building blocks of the project. Their interaction is illustrated in the system architecture presented in Section 3.1 by losanges.

- LP Optimization Solver (DCOPF)
- Data Ingestion & Preprocessing
- Investment Metrics calculation
- AI-based reporting

Each element addresses a distinct requirement, from solving the DC power flow problem to generating final scenario reports with optional AI-driven commentary.

3.2.1 Data Ingestion & Preprocessing

Data Handling
Data handling follows a three-tiered structure:

1. `data/raw`: Unaltered sources such as annual wind/solar profiles from public databases or raw load curves provided by our supervising professor.
2. `data/processed`: Intermediate files that have undergone partial cleaning (e.g., timestamps alignment, filtering outliers). As part of this step, we also perform a *Seasonal and Trend decomposition using Loess (STL)* to identify anomalies in the load profile, following methods described in [2]. The seasonal median week was picked for each season.
3. `data/working`: used .csv files directly by the solver and scenario scripts (`master_gen.csv`, `master_load.csv`). These files are concise, containing only the time-series data and parameters needed for each scenario run.

The STL decomposition applied during the preprocessing step to validate our *typical-week* selection for each season (Figure 6). The data showed a broad U-shaped trend (lower demand in warmer months) and strong daily/weekly seasonality. Residuals exhibited higher variance at the start and end of the year, suggesting possible holiday or extreme-weather anomalies. Excluding those outlier weeks helped ensure our final “median” week captures typical load patterns.

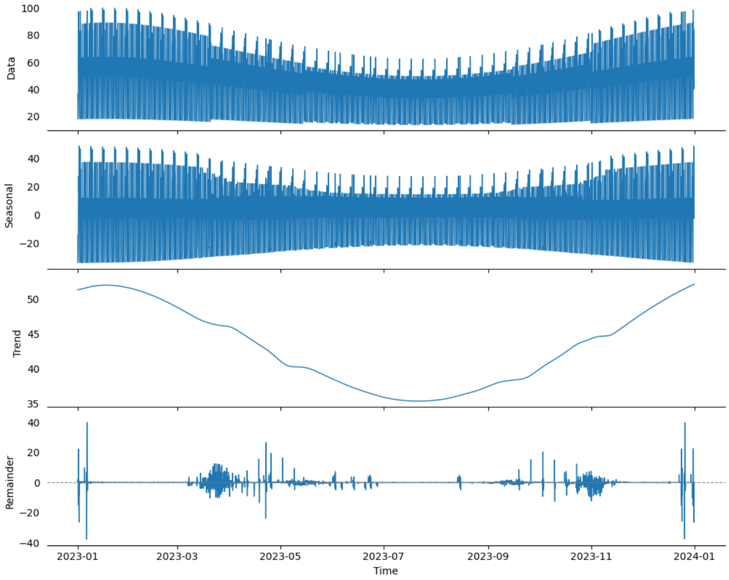


Figure 6: STL decomposition of the annual load data

Scripts for Data Preprocessing
Two Python scripts form the backbone of our data preprocessing:

- `create_master_gen.py`: Consolidates multiple raw generation sources (wind, solar, nuclear, etc.) into a unified time series, selecting a *median week* per season to reduce hours from 8,760 to just 7×24.
- `create_master_load.py`: Builds coherent load profiles for each season, optionally shifting one bus's demand by a week if needed. Buses 5 and 6 serve as the primary load centers in our example network.

These scripts output `master_gen.csv` and `master_load.csv` in `data/working`. By scaling each typical week by 13 (winter/summer) or 26 (spring/autumn), the method preserves core weekly cycles—consistent with the STL findings—while excluding outlier periods. This provides a robust yet computationally efficient basis for annual cost estimation.

Network Modeling
A simplified network is defined via `branch.csv` and `bus.csv` in `data/working`. Key characteristics of this grid (e.g., line impedances, bus voltage levels) are presented in Section 4. While our example focuses on a small test system, larger networks (e.g., IEEE 30-bus or 118-bus) can be incorporated using the same file structure.

3.2.2 LP Optimization Solver

In this module, the `dcopf.py` script translates the DC power flow problem and associated operational constraints into a linear program (LP). Although the mathematical formulation (objective function and constraints) has been fully described in Section 2, we outline here how these elements are *implemented* at the code level, focusing on the handling of hourly dispatch and storage dynamics.

Time-Step Stacking
For each hour t of the representative dataset (e.g., 24 hours $\times$ 7 days per season), the solver creates:

- **Generation Variables**
  It ensures the solver respects asset-level capacity limits in every hour. In `dcopf.py`, each dispatchable asset g is assigned a variable `GEN[g, t]` that ranges between `pmin` and `pmax`. For each time step t, we create:

```python
for g in G:
  for t in T:
      pmin = ...
      pmax = ...
      GEN[g, t] = pulp.LpVariable(
          f"GEN_{g}_{t}_var",
          lowBound=pmin, upBound=pmax)
```

- **Voltage Angles** (Phase Angles) at Each Bus
  The DC power flow constraints then enforce line flows based on the angle difference between connected buses. A dictionary `THETA[i, t]` stores the phase angle at bus i and time t:

```python
for i in N:      # set of buses
  for t in T:    # set of time steps
      THETA[i, t] = pulp.LpVariable(
          f"THETA_{i}_{t}_var", lowBound=None, upBound=None)
```

- **Line Flow Variables**
  For each branch (i, j) in `branch.csv`, we create `FLOW[i, j, t]`, constrained by the line's thermal limit (`rateA`). The relevant code snippet looks like:

```python
for idx, row in branch.iterrows():
  i = int(row['fbus'])
  j = int(row['tbus'])
  limit = row['ratea']
  for t in T:
      FLOW[i, j, t] = pulp.LpVariable(
          f"FLOW_{i}_{j}_{t}_var",
          lowBound=-limit, upBound=limit)
```

  Here, negative values indicate flow in the opposite direction, respecting the bidirectional capacity of AC transmission lines (under the DC approximation).

These variables are repeated across all time steps, thereby stacking the corresponding constraints to capture the evolution of dispatch decisions over the week.

Storage Modeling
A key feature in `dcopf.py` is the *storage* handling. The code introduces:

- **Charge/Discharge Variables** for each storage asset, bounded by $\pm P_{\max}$. Each storage asset s has two distinct variables: `P_charge[s, t]` (charge rate) and `P_discharge[s, t]` (discharge rate), both bounded by $\pm P_{\max}$. Below is a simplified Python excerpt:

```python
for s in S:  # S is the set of storage IDs
    # Retrieve max power from CSV or data context
    P_max = storage_info[s]['pmax']
    for t in T:
        P_charge[s, t] = pulp.LpVariable(
            f"P_charge_{s}_{t}",
            lowBound=0,   # cannot be negative
            upBound=P_max)
        P_discharge[s, t] = pulp.LpVariable(
            f"P_discharge_{s}_{t}",
            lowBound=0,   # cannot be negative
            upBound=P_max)
```

Note that charge and discharge are individually constrained to be non-negative, ensuring the net power from storage ($P_{\text{discharge}} - P_{\text{charge}}$) stays within $\pm P_{\max}$.

- **State of Charge (SoC) Variables:**
The SoC at each time step $t+1$ depends on the SoC at time t, as well as any charge or discharge volumes. To ensure continuity, we define an *extended time index* such that $t_0 = t_{24\times 7+1}$, creating a cyclical boundary condition where the end-of-horizon SoC equals the initial one.

Objective Function & Solver Execution.
The objective function *sums the dispatch costs* of all generators over each time step. After constructing these LP constraints, the script invokes PuLP's interface to the CBC solver to:

1. **Assemble** the problem (variables, constraints, and objective) in standard form:

$$\begin{aligned} \min_{\mathbf{x}} \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{aligned}$$

where $\mathbf{x}$ contains all decision variables (generation, flows, storage), $\mathbf{c}$ represents costs, $\mathbf{A}$ encodes network constraints, and $\mathbf{l}, \mathbf{u}$ are variable bounds.

2. **Solve** for an *hourly dispatch schedule* that minimizes total cost while respecting line flows and operational limits.

3. **Extract** the final solutions (e.g., `gen`, `flow`, `storage` states), storing them in data frames.

Scaling to Annual Results.
Because each run typically focuses on a single "typical week" per season, the corresponding cost is subsequently scaled (see Section 3.2.1) to approximate annual figures. While this reduces the computational load by omitting all annual 8,760 hours, computing 3 seperaed weeks of 504 hours (24 hours $\times$ 7 days) instead. It preserves the essential behavior of generation and storage dispatch.

3.2.3 Financial Metrics Module

Investment decisions in infrastructure oftens rely on 3 key aspects : numbers (metrics), interest rates i , and time horizon T. The calculations are performed in `create_master_invest.py`.

Interest rates for electrical infrastructure typically range from 5% to 10% [4] Its value has significant impact on the calculations and, consequently, the investment decision. Time horizon often depends on the project's expected lifetime, but can also be subject to regulatory constraints. For metrics, we can rely on the Net Present Value (NPV) and the annuity (a).

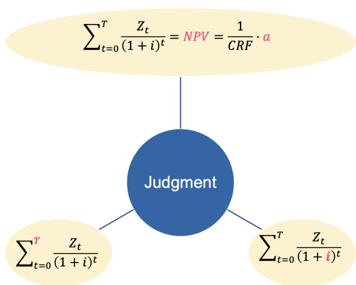


Figure 7: The three judgement dimensions influencing investment decisions by T. Herrmann

Net Present Value (NPV)
The NPV sums the present value of all cash flows over the asset lifetime. If positive, it is profitable.

$$\text{NPV} = Z_0 + \sum_{t=0}^{T} \frac{Z_t}{(1+i)^t} \tag{13}$$

where Z_0 represents the initial investment cost at $t = 0$ and Z_t are the cash flows (costs or revenues) in period t.

By repeating the NPV calculation for each scenario, we can compare the financial viability of different investment options. The scenario with the highest positive NPV is the most profitable one.

Annuity
An other dynamyic decison making metric is the *annuity* which permits to assess decison making on assets with different lifetimes. It denotes an equivalent constant monetary input over the period under consideration.

Is calculated calculated from the NPV by multiplying it by the annuity factor (Capital Recovery Factor, CRF) :

$$a = NPV \cdot \frac{i(1+i)^T}{(1+i)^T - 1} = NPV \cdot CRF \tag{14}$$

Sensitivity Analysis
While sensitivity analyses typically focus on varying discount rates, we instead analyze load variations ($\pm 20\%$) since interest rates were already discussed in Section 3.2.1. This helps assess both investment robustness, under demand uncertainty and identify potential risk of network constraints requiring additional capacity.

3.2.4 Automatic AI-Based Reporting

This module, `scenario_critic.py`, integrates scenario data (annual costs, generator dispatch, net present values, etc.) with an AI-based module to automatically produce Markdown reports for each scenario. Concretely, it performs:

1. **API Integration**
   A class (`ScenarioCritic`) initializes an OpenAI client using an API key [7]. We define a "context prompt" describing the energy system mix (nuclear, gas, wind, etc.), relevant cost metrics, and storage assets.

2. **Generating a Critical Analysis**
   The script collects scenario outputs (e.g. annual_cost, generation by asset, etc.) into a short "user" prompt. It requests an AI-generated critique. We chose to focus on the economic efficiency, strengths/weaknesses, and possible improvements. This feedback is concise (up to 200 words) and helps users quickly assess each scenario.

3. **Automatic Markdown Reports.**
   Using both values from the optimization/investment metrics module and the AI critique, the script compiles a scenario-specific `.md` file (e.g., `scenario_1_analysis.md`). An example of the reports can be found in the Appendix. The final report includes:

   - *Key Financial Metrics*: Initial investment, annual cost, various NPVs.
   - *Generation Statistics*: Generation costs by asset, capacity factors, seasonal trends.
   - *AI Critical Analysis*: The generated text is appended to the report's conclusion, providing a quick executive-level summary.
   - *Plots and Figures*: If configured, the script references or embeds seasonal generation comparisons and annual summaries.

Cost Analysis
It is important to note that the OpenAI API has a cost – 0.00015 USD per token. As seen in the Figure below on the 29th of January 2025, for our 12 scenarios plus 1 global (summary) report, it costed us $0.0035 USD (± $0.00027 per scenario). While negligible compared to the investment decisions analyzed sums, this is not a free feature. Other open-source LLMs usage could be evaluated.

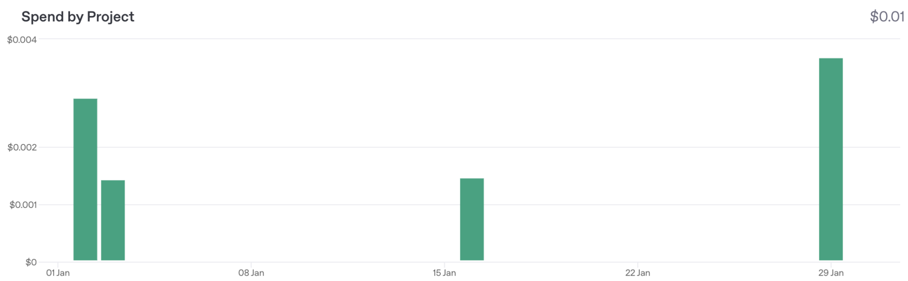


Figure 8: January 2025 project cost chart of OpenAI API Usage (Report Generation only)

The AI analysis feature was implemented as an optional component that can be enabled or disabled according to user preference prior to running the optimization process. Hence guaranteeing that the platform can be used by users completely free of costs.

3.3 Technical Implementation

This section covers the practical aspects of using Python and related libraries for solving the DCOPF and generating scenario results. It also addresses performance considerations.

3.3.1 Programming Language & Libraries

All scripts are written in Python 3.10, leveraging the following key libraries:

- `pulp` for linear programming and interfacing with `CBC` [3]. The verssion used is 2.9.0.
- `pandas`, `numpy`, and `matplotlib/seaborn` for data manipulation and visualization.
- `networkx` for optional graph-based analyses (e.g., if we extend to network exploration).
- `openai` for LLM access and generation of reports.

It was decided to use Poetry for dependency management and packaging, ensuring consistent versions across different environments. A detail .toml listing all dependencies is available if needed. Poetry facilitates the containerization of the platform.

While parts of the project are designed for containerization, certain components like the OpenAI API key require secure handling of personal credentials. Currently, the project runs locally without containerization, though its architecture was developed with it in mind.

3.3.2 Performance

While the solver has successfully handled a handful of scenarios (e.g., 20–40), it has yet to be benchmarked against large-scale or more complex networks. As order of magnitude, we computed different scenarios including sensitivity analysis and scenario plot generation enabled. No AI report generation was enabled. We yield a linear correlation :

As this was my first major programming project beyond small scripts, I prioritized a flexible, readable codebase over computational efficiency. Nonetheless, handling hourly time-series data for an entire

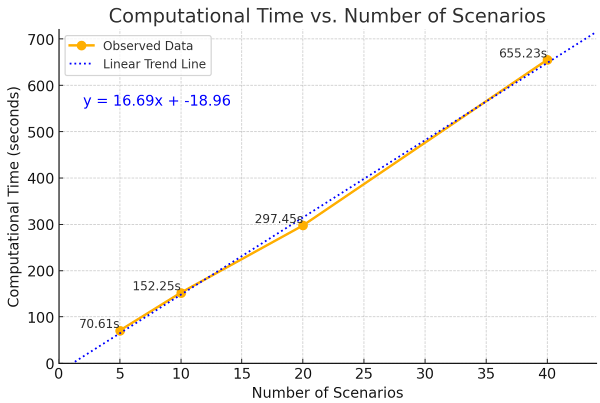


Figure 9: Computation time vs number of scenarios (correlation for limited number of scenarios only)

year inherently creates a large number of constraints (over 8,700). Real-world systems even adopt finer temporal resolutions (e.g., 15-minute intervals).

To keep run times feasible, we selected representative weeks per season—an approach that cuts computing by roughly 17x. (With 52 weeks in a year reduced to just 3 representative weeks – $52/3 = 17$).

4 Results and Validation

4.1 Test Cases (Scenario Definitions)

4.1.1 Configuration

We used a simple 9-bus network topology, as illustrated below.

While the initial design considered multiple load buses (at least two), technical implementations limitations in integrating storage units with multiple load centers led us to simplify the model to use only bus N°5 as a load bus. The remaining buses are available for scenario-specific generation placement. The network consists of standardized transmission elements with the following characteristics:

- Uniform line parameters: r = 0.00281 p.u., x = 0.0281 p.u., b = 0.00712 p.u.
- Consistent thermal limits: 60 MW for all branches
- Voltage bounds: $0.9 \leq V \leq 1.1$ p.u.
- Base voltage level: 230 kV

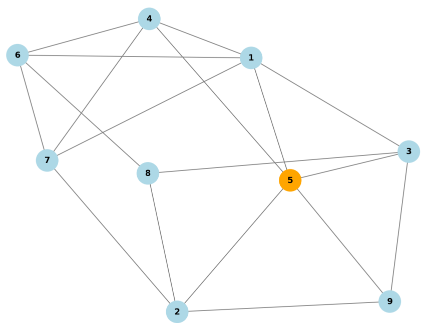


Figure 10: Base topology network

To systematically analyze different network configurations, we defined a simple population of 10 scenarios for the sake of clarity in this report. These scenarios explore various combinations of generation and storage placements within our pre-defined network. Each scenario specifies:

1. The location and type of generation units (nuclear, solar, wind, or gas) at different buses
2. The placement of storage units (Battery1 or Battery2) if any
3. A load factor to scale the demand - in our case kept by default at 1.0 for our analysis.

Scen.	Generation Positions	Storage Units
1	{Bus 1: Nuclear, Bus 4: Solar}	*None*
2	{Bus 1: Nuclear, Bus 4: Solar}	{Bus 2: Bat1, Bus 7: Bat1}
3	{Bus 1: Nuclear, Bus 4: Solar, Bus 2: Wind}	{Bus 7: Bat1}
4	{Bus 2: Nuclear, Bus 4: Wind, Bus 8: Gas}	{Bus 1: Bat1, Bus 7: Bat2}
5	{Bus 1: Wind, Bus 2: Solar, Bus 3: Nuclear}	{Bus 8: Bat1, Bus 4: Bat2}
6	{Bus 2: Nuclear, Bus 4: Wind, Bus 7: Solar}	*None*
7	{Bus 3: Solar, Bus 4: Nuclear, Bus 8: Wind}	{Bus 1: Bat2}
8	{Bus 1: Gas, Bus 4: Nuclear, Bus 7: Solar}	{Bus 9: Bat2, Bus 2: Bat2}
9	{Bus 2: Wind, Bus 4: Nuclear, Bus 7: Solar, Bus 1: Solar}	*None*
10	{Bus 2: Wind, Bus 4: Nuclear, Bus 7: Solar, Bus 1: Solar}	{Bus 3: Bat2, Bus 9: Bat2}

Table 1: 10 scenarios case-study from `scenarios_parameters.csv`

4.1.2 Data

The generation units have distinct characteristics – static limits and variable generation profiles. Constant limits such as nuclear power plants and gas turbines have a fixed power output. Their costs and max power were defined with dummy variables. Solar and wind are variable and depend on weather patterns hence have a variable availability profile.

Type	P_{max} (MW)	Cost ($/MWh)
Nuclear	800	5.0
Gas	250	8.0
Wind	Variable	0
Solar	Variable	0

Table 2: Generation Unit Specifications

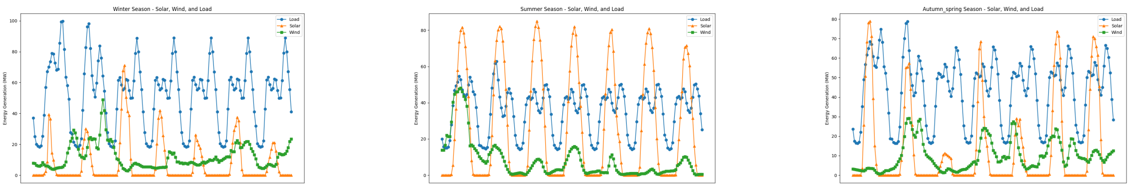
Type	Power (MW)	Capacity (MWh)	Efficiency
Battery1	±30	60	99%
Battery2	±55	110	99%

Table 3: Storage Unit Specifications

The load who can be interpreted as the demand of a little town (max. 100MW) also is variable. This hourly fluctuating data was obtained from different sources :

- The load profile was provided by the supervising professor. An STL decomposition was applied to confirm the recurring weekly patterns and seasonalities.
- The renewable generation data was obtained from renewables.ninja [8] for a location in Sion, Switzerland (46.231°N, 7.359°E). Both solar PV and wind configurations were then processed and scaled.

Their profiles can be observed below in their respective seasonal weeks. As discussed in Section 3.2.1, these week choices were made based on the mean load demand per week in their respective season. We aimed for the residuals to be normally distributed in these chosen weeks.



(a) Winter (WK 02) (b) Summer (WK 31) (c) Spring/Autumn (WK 43)

Figure 11: Weekly seasonal load and generation availability profiles

4.2 Operational Results

4.2.1 Hourly Dispatch

Below is an example of an hourly dispatch during a summer week. As expected it shows some 24-hour seasonality in both the generation and demand. The stacked area plot reveals a clear daily pattern where solar generation (dashed orange line) peaks during midday hours while wind generation (dashed dark blue line) provides more variable output throughout the day. The total generation profile closely follows but slightly exceeds the demand curve (black line), with the excess being stored in batteries for later use.

We notice that during peak solar hours, when photovoltaic output exceeds demand, the surplus free of cost energy is stored in the battery. This stored energy is then discharged during evening hours when solar generation declines but demand remains high – idem with the wind generation. While more difficult to notice in this configuration, the batteries discharge at the end of the calculation cycle to meet the condition $SOC_0 = SOC_{final}$.

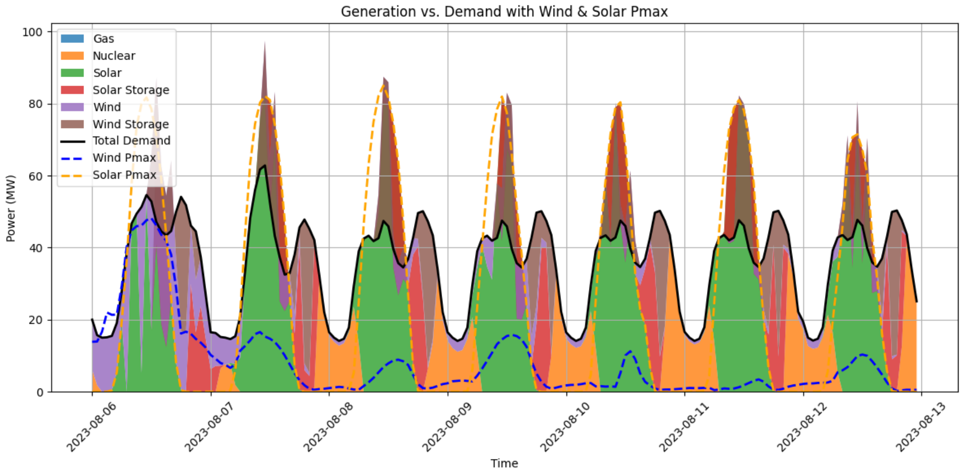
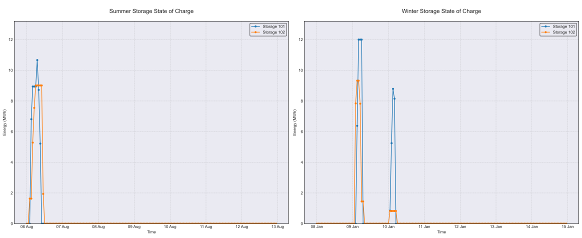


Figure 12: Hourly dispatch for Scenario 5

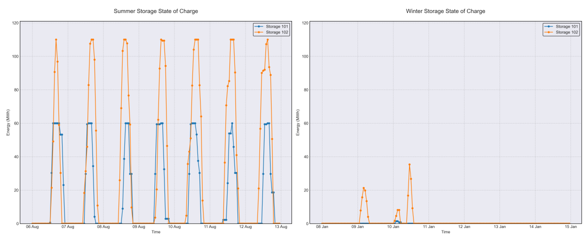
On day 2, total generation exceeds demand likely due to the battery model allowing simultaneous charging and discharging. Without constraints preventing concurrent charge/discharge operations, the solver can exceed single-direction power limits by setting both P_charge and P_discharge nonzero in the same hour.

4.2.2 Feasibility & Technical Observations

To validate the model's behavior, particularly after simplifying from multiple loads to a single load bus, we analyzed the binding constraints in two contrasting scenarios (4 and 5). A review of the solver logs in `dcopf.py` revealed several critical binding constraints that help verify proper system operation:



(a) Scenario 4, SoC left: Summer, right: Winter



(b) Scenario 5, SoC left: Summer, right: Winter

Figure 13: State of Charge comparison between Scenarios 4 and 5

The State of Charge (SOC) comparison between Scenarios 4 and 5 (Fig. 13) reveals striking differences in storage utilization patterns, despite both scenarios having identical generation and storage capacities. The key distinction lies in the generation type at Bus 2:

- In Scenario 4, with nuclear generation at Bus 2, we observe relatively modest SOC variations. This reflects the steady, baseload of nuclear power,(consistent output regardless of time of day). We can assume that the batteries primarily serve to optimize power flow rather than accommodate large generation swings. They peaked on their only first usage day suggesting an emerging need for storage (60 MWh x2)
- Scenario 5, featuring solar generation at Bus 2, shows much more SoC fluctuations. The pronounced midday solar peaks drive rapid battery charging– seeing them reaching their limit every day suggesting a size increase. While evening hours see significant discharge as stored energy supplements the diminished solar output.On the other hand, in summer weeks the batteries are not used at all.

The network topology, particularly the lines connecting Bus 2 to buses 5, 7, 8, and 9, plays a key role in distributing generation and enabling storage utilization. The placement of generation units impacts power flows and storage patterns throughout the network.

4.2.3 Generation dispatch

Line flow analysis during peak hours would likely show congestion on transmission corridors connecting major generation sources to Bus 5. While detailed hourly patterns weren't examined, the seasonal-week models explored. In the context of "investment" analysis, stakeholders would likely be interested in the global generation dispatch and their associated trends.

As expected, nuclear generation increases significantly during winter periods to meet higher demand. Most notably in scenario 5, solar generation reaches impressive levels that even exceed nuclear output during peak periods, suggesting the potential for solar to serve as a primary generation source.

The annual generation profiles shown below illustrate the key seasonal patterns and generation mix across these two scenarios:

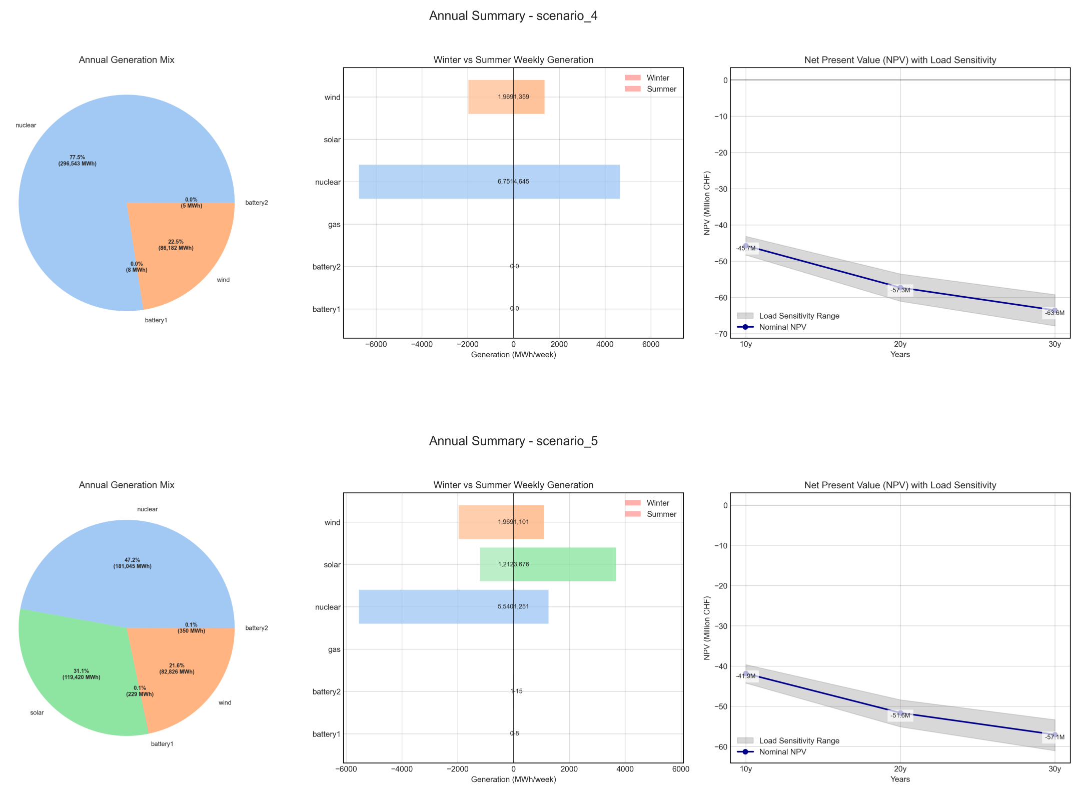


Figure 14: Summary plots of annual generation profiles for investment reports

In Scenario 5, solar generation reaches impressive levels that even exceed nuclear output during peak periods. This scenario demonstrates how zero-cost renewable generation can create significant economic advantages. Scenario 5 ranks second-best in overall costs whereas Scenario 4's conventional generation mix shows the worst cost performance, highlighting the financial benefits of integrating renewable resources into the power system.

4.3 Investment & Economic Analysis

The economic assessment focuses exclusively on renewable energy and storage assets, as these represent the key investment decisions. While nuclear and gas generation costs are modeled in the optimization to determine optimal dispatch, they are excluded from the investment analysis since our primary goal is to evaluate the financial viability of green infrastructure additions to an existing conventional generation fleet.

The following parameters were used:

Parameter Type	**Technology**	**Value**
CAPEX (CHF/MW)	Wind	1'400'000
	Solar	900'000
	Battery Type 1	250'000
	Battery Type 2	450'000
Technical Lifetime (years)	Wind	19
	Solar	25
	Battery Type 1	6
	Battery Type 2	8
Annual OPEX (% of CAPEX)	Wind	4%
	Solar	2%
	Battery Type 1	3%
	Battery Type 2	4%

Table 4: Updated financial and technical parameters used in the analysis

The results illustrate a versatile platform that integrates various generation and storage technologies, each contributing unique cost and performance profiles. Renewable assets like wind and solar offer attractive CAPEX values, while their associated OPEX and technical lifetimes shape long-term economics, particularly when combined with battery systems that require more frequent replacement.

Among the scenarios analyzed, the mix in Scenario 7, which includes solar, nuclear, and wind generation with a single Battery Type 2, delivers the lowest annualized cost and most favorable net present values over 10 and 30 years. This example highlights the potential of combining diverse assets to achieve a balanced and economically sustainable energy mix.

Conversely, the configuration in Scenario 4, featuring nuclear, wind, and gas with two types of battery storage, demonstrates how additional storage and more complex asset mixes can elevate both upfront and recurring costs. This underlines the importance of optimizing asset selection and mix based on specific case study requirements.

Overall, these results serve as a foundational example, showcasing the platform's capability to compare different asset combinations. The insights gained here can be further refined for tailored applications in specific case studies, emphasizing the critical role of both capital investment and long-term operational considerations in energy system planning.

Scen.	**Initial Inv.**	**Annual Cost**	**10y NPV**	**30y NPV**	**Annuity**
7	2'750'000	942'766	-9'918'474	-15'233'659	1'353'167
3	2'550'000	980'376	-9'821'150	-15'258'653	1'355'387
6	2'300'000	1'048'043	-9'829'002	-15'353'770	1'363'836
5	3'000'000	905'287	-10'113'189	-15'478'398	1'374'906
9	3'200'000	1'048'043	-10'849'783	-16'578'091	1'472'589
1	900'000	1'380'923	-10'286'888	-16'770'455	1'489'676
10	4'100'000	887'122	-11'361'773	-16'896'633	1'500'885
2	1'400'000	1'300'109	-10'637'014	-17'193'991	1'527'298
8	1'800'000	1'275'673	-11'172'440	-17'715'738	1'573'644
4	2'100'000	1'482'721	-12'967'033	-20'754'695	1'843'586

Table 5: Ranked by annuity, financial comparison across scenarios

4.3.1 Sensitivity Analysis

We evaluated how a ±20% variation in load affects each scenario's NPV over time. The table shows that scenarios with high gas dependency (e.g., Scenario 4) exhibit greater NPV volatility compared to renewable-heavy configurations. A 20% load increase causes Scenario 4's 30-year NPV to deteriorate by 58%, while Scenario 5's more diverse mix limits the impact to 75%:

Table 6: NPV Analysis for Scenarios 4–5 under Load Variations (10, 20, 30-year horizons)

Scenario	**Load Factor**	**NPV (10yr)**	**NPV (20yr)**	**NPV (30yr)**
Scenario 4	0.8	-11'172'440	-15'548'394	-17'715'738
	1.0	-12'967'033	-18'400'575	-20'754'695
	1.2	-14'761'626	-21'252'756	-23'793'652
Scenario 5	0.8	-8'090'551	-11'046'381	-12'382'718
	1.0	-10'113'189	-13'807'976	-15'478'398
	1.2	-12'135'827	-16'569'571	-18'574'078

We can conclude that the sensitivity analysis provides insights into resilience of the scenarios.

4.4 Report Generation and Insights

The AI report generates a summary of the results of individuals scenarios focus on three aspects for the individual scenario :

- Economic Efficiency of the Generation Mix
- System Composition Strengths/Weaknesses
- Key Recommendations for Improvement

while the global report provides a comprehensive overview of all scenarios, highlighting the optimal and suboptimal scenarios. It focus on the following aspects :

- Overall Trends in Cost Effectiveness
- Trade-offs Between Different Generation Mixes
- Key Success Factors in Better Performing Scenarios
- Recommendations for Future Scenario Design

In the individual report, while the AI effectively identifies scenarios and references analytical data, its descriptions remain somewhat generic and lack the depth of expert analysis. In its current configuration, it serves as a useful complement to, rather than replacement for experts advice. The strength lies in providing a context-based overview that helps investors quickly grasp the key implications of each scenario.

Also, no specific prompt engineering was performed to optimize the handling of metrics. With a more tailored prompt and detailed context and objectives, such as battery dimensioning or investment thresholds, the AI could generate more targeted and insightful reports that better serve decision-making purposes.

However, the AI's global summary report effectively identifies optimal and suboptimal scenarios while providing comprehensive comparisons across multiple evaluation criteria – on a superior level than the scenario-based report. The AI integration proves particularly valuable in generating regression analyses and uncovering relationships between predictor variables, leading to meaningful insights of assets usage and their associated costs. The complete global summary is available in the appendix.

5 Discussion

5.1 Limitations

Multiple Load Profiles. Although the load profiles used in the test case are static (their buses is pre-defined) by platform design it is often the case in real life. However, we did not handle multiple loads at once, and the platform is not yet designed to handle them. This is a limitation that should be addressed in future work. It suggests is yet designed as a single demand per bus-time pair.

Limited Benchmarking. The main limitation of the platform is that it lacks comprehensive benchmarking against established standards or similar systems. While the platform demonstrates functionality, there is no systematic comparison (e.g. IEEE test cases). This makes it difficult to objectively assess the platform's effectiveness compared to existing solutions.

5.2 Methodological Insights

Technical-Economic Integration. The platform's key methodological contribution lies in establishing a direct link between technical system design and economic valuation. By integrating technical dispatch optimization with comprehensive financial analysis, it enables stakeholders to evaluate both the operational feasibility and economic viability simultaneously during the design phase. This represents a significant advancement over traditional approaches that often treat technical and economic assessments as separate processes.

Renewable Asset Valuation. The optimization framework enables assessment of renewable energy assets by quantifying the costs of conventional "dirty" generation within the entire system. This cost metric, when combined with investment parameters, provides a clear basis for comparing different scenarios and determining the profitability of renewable alternatives. Rather than evaluating green assets in isolation, this approach reveals their true economic value by measuring their impact on overall system costs.

Storage Optimization. The platform's analysis of battery state-of-charge patterns reveals opportunities for optimizing storage deployment timing and sizing. By examining partial-year installation scenarios, as shown in the state-of-charge comparison figures, we can better determine not just the optimal storage capacity but also when during the year new storage should be commissioned. This temporal dimension of storage deployment represents an important area for future framework development.

Maintenance Management. The platform could be enhanced by incorporating a maintenance rerouting system that creates temporary alternative paths during asset downtime. This would allow for seamless transitions during maintenance periods by automatically redirecting power flows through equivalent backup systems. Such functionality would improve system reliability and provide more realistic operational scenarios that account for planned and unplanned maintenance events.

5.3 Future Directions

Benchmarking and Validation. A critical next step is establishing comprehensive benchmarking against industry standards and IEEE test cases. This would validate the platform's performance and provide objective comparisons with existing solutions, building confidence in its results and highlighting areas for improvement.

Real-Time Price Integration. Incorporating real-time electricity pricing mechanisms would significantly enhance the platform's practical utility. This could enable:

- Analysis of arbitrage opportunities between peak and off-peak periods
- Evaluation of green energy resale potential, particularly for private owners
- More accurate modeling of revenue streams from grid services
- Dynamic optimization of storage charging/discharging based on market conditions

Code Optimization. While computational efficiency is not an immediate concern, for any serious use of the platform the code reliability and maintainability should be improved. Same goes for the interactions between the different components.

Enhanced Prompt Engineering. Further prompt engineering work could improve report readability, aiming at facilitating stakeholder engagement. This would ensure key metrics and findings are presented in clear, actionable formats aligned with industry standards.

5.4 Concluding Remarks

This platform successfully bridges technical dispatch simulations with investment analysis in energy systems. The test cases validate the core functionality while demonstrating how sensitivity analyses, scenario comparisons, and AI-enhanced reporting can generate actionable insights. Future development priorities should be chosen based on specific use cases, whether utility-scale planning, microgrid optimization, or renewable integration. However, benchmarking against industry standards and IEEE test cases is a must.

6 Conclusion

This project has successfully developed and demonstrated a comprehensive investment model for optimizing technology asset deployment in integrated energy systems. The platform combines DC Optimal Power Flow (DCOPF) simulations with detailed investment analysis, providing valuable insights for decision-makers. Key achievements and conclusions include:

- **Investment-Focused Optimization:** The platform effectively evaluates different technology combinations through both technical and economic lenses. As demonstrated in the 9-bus case study, it can simultaneously assess multiple aspects:
  - Capital expenditure impacts on long-term profitability
  - Operational costs and their influence on Net Present Value
  - Storage sizing and its role in system economics
  - Trade-offs between conventional and renewable technologies
- **Economic Analysis Capabilities:** The model provides robust financial metrics for decision-making:
  - Net Present Value (NPV) calculations across different time horizons (10, 20, 30 years)
  - Annuity comparisons for assets with different lifetimes
  - Sensitivity analysis to evaluate investment robustness and simulate load increase

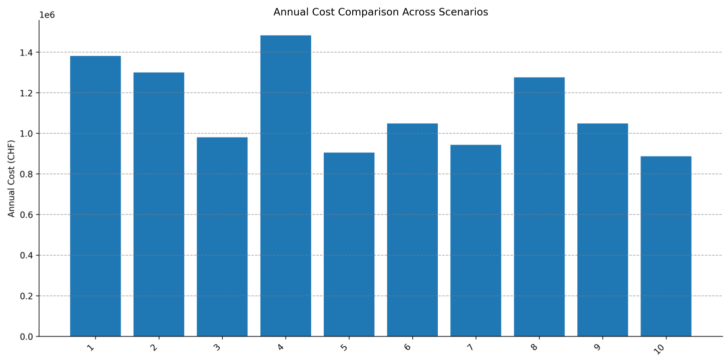


Figure 15: Annuity comparison across scenarios

As shown in Figure 15, scenarios with balanced technology mixes (e.g., Scenarios 7 and 5) achieved the lowest annuities, around 1.35M CHF/year. These configurations with renewable generation and appropriate storage, demonstrated the value of diversified technology portfolios.

In contrast, scenarios heavily dependent on gas generation or oversized storage (e.g., Scenario 4) showed significantly higher annuities, reaching 1.84M CHF/year.

From a technical standpoint, the platform integrates DCOPF network constraints, multiple energy carriers, storage dynamics, and renewable generation variability. And, from a decision support standpoint, the system enables rapid scenario comparison, identifies cost drivers, assesses investment risks, and provides AI-enhanced analysis.

While the case study utilized a simplified 9-bus system, the methodology and platform have demonstrated their capability to handle the core requirements of investment decision support in energy systems. The results show that optimal technology selection depends on multiple factors, including:

- Initial investment constraints
- Operational cost considerations

- Technology lifetime and replacement cycles
- System reliability requirements

Future development should be guided by use case requirements first. For utility-scale planning, priorities may include multi-carrier integration (gas, hydrogen) and sophisticated environmental assessments. For microgrid optimization, the focus should be on real-time pricing and enhanced storage modeling. For optimal energy system dimensioning, improving the framework to better determine optimal generator sizes based on demand profiles would be valuable. For renewable integration studies, improving sensitivity analysis capabilities and AI-driven scenario evaluation would be most valuable.

Rather than pursuing all improvements simultaneously, development efforts should align with the intended application to ensure appropriate depth and accuracy where it matters most.The platform provides a solid foundation for investment decision-making in energy systems, balancing technical feasibility with economic viability. Its modular architecture allows for future extensions to address more complex scenarios and additional optimization objectives, such as emissions reduction, while maintaining its core strength in economic assessment and technology selection.

Acknowledgements

For the redaction of this report, I would like to acknowledge the use of artificial intelligence to improve the clarity and structure of my sentences. The core observations, analyses, and personal reflections are entirely my own, drawn from my experiences during the field trip and subsequent research. The LLM usage was employed primarily for language refinement, code formatting, and orthographic corrections. Its integration helped communicate complex concepts clearly and effectively.

A Code Model

A.1 Optimization Model

```python
def dcopf(gen_time_series, branch, bus, demand_time_series, delta_t=1):
    import pulp
    from pandas.tseries.offsets import DateOffset
    import numpy as np
    import pandas as pd
    import math

    print("[DCOPF] Entering dcopf function...")
    print(f"[DCOPF] gen_time_series length = {len(gen_time_series)}, demand_time_series length = {len(demand_time_series)}")

    # Create LP problem
    DCOPF = pulp.LpProblem("DCOPF", pulp.LpMinimize)

    # Identify storage vs. non-storage units
    storage_data = gen_time_series[gen_time_series['emax'] > 0]
    S = storage_data['id'].unique()     # set of storage IDs
    non_storage_data = gen_time_series[gen_time_series['emax'] == 0]
    G = non_storage_data['id'].unique() # set of non-storage gen IDs

    print(f"[DCOPF] Found storage units: {S}, non-storage units: {G}")
    print("[DCOPF] Storage data sample:")
    print(storage_data[['id', 'bus', 'emax', 'pmax', 'eta']].head())

    # Time and bus sets
    N = bus['bus_i'].values
    T = sorted(demand_time_series['time'].unique())
    if not T:
        print("[DCOPF] No time steps found in demand_time_series. Returning None.")
        return None

    next_time = T[-1] + DateOffset(hours=delta_t)
    extended_T = list(T) + [next_time]

    # 1. Create GEN variables for non-storage generators
    GEN = {}
    for g in G:
        gen_rows = gen_time_series[gen_time_series['id'] == g]

        # We assume one row per time step or time-invariant parameters
        for t in T:
            row_t = gen_rows.loc[gen_rows['time'] == t]
            if row_t.empty:
                print(f"[DCOPF] Missing data for generator={g}, time={t}. Returning None.")
                return None

            pmin = row_t['pmin'].iloc[0]
            pmax = row_t['pmax'].iloc[0]
            GEN[g, t] = pulp.LpVariable(f"GEN_{g}_{t}_var", lowBound=pmin, upBound=pmax)

    # 2. Voltage angle variables
    THETA = {
        (i, t): pulp.LpVariable(f"THETA_{i}_{t}_var", lowBound=None)
        for i in N for t in T
```

```python
    }

    # 3. FLOW variables
    FLOW = {}
    for idx, row_b in branch.iterrows():
        i = int(row_b['fbus'])
        j = int(row_b['tbus'])
        for t in T:
            FLOW[i, j, t] = pulp.LpVariable(f"FLOW_{i}_{j}_{t}_var", lowBound=None)

    # 4. DC Power Flow Constraints
    for idx_b, row_b in branch.iterrows():
        i = int(row_b['fbus'])
        j = int(row_b['tbus'])
        susceptance = row_b['sus']

        for t in T:
            DCOPF += FLOW[i, j, t] == susceptance * (THETA[i, t] - THETA[j, t]), \
                     f"Flow_Constraint_{i}_{j}_Time_{t}"

    # 5. Storage Variables/Constraints
    P_charge = {}
    P_discharge = {}
    E = {}

    for s in S:
        s_row = gen_time_series.loc[gen_time_series['id'] == s].iloc[0]
        E_max = s_row['emax']
        E_initial = s_row['einitial']
        eta = s_row['eta']
        P_max = s_row['pmax']  # Discharging max

        # Create charge/discharge variables
        for t in T:
            P_charge[s, t] = pulp.LpVariable(f"P_charge_{s}_{t}_var", lowBound=0, upBound=abs(P_max))
            P_discharge[s, t] = pulp.LpVariable(f"P_discharge_{s}_{t}_var", lowBound=0, upBound=abs(P_max))

        # SoC variables
        for t in extended_T:
            E[s, t] = pulp.LpVariable(f"E_{s}_{t}_var", lowBound=0, upBound=E_max)

        # Initial SoC
        DCOPF += E[s, T[0]] == E_initial, f"Initial_Storage_SoC_{s}"

        # SoC dynamics: E[next] = E[t] + eta*charge - (1/eta)*discharge
        for idx_t, t in enumerate(T):
            next_t = extended_T[idx_t + 1]
            DCOPF += E[s, next_t] == E[s, t] + eta * P_charge[s, t] * delta_t - (1/eta) * P_discharge[s, t] * delta_t, \
                     f"Storage_Dynamics_{s}_Time_{t}"

        # Final SoC (optional)
        DCOPF += E[s, extended_T[-1]] == E_initial, f"Final_Storage_SoC_{s}"

    # 6. Slack bus angle = 0
    slack_bus = 1  # Adding back the slack bus constraint
    for t in T:
        DCOPF += THETA[slack_bus, t] == 0, f"Slack_Bus_Angle_Time_{t}"
```

```

    # 7. Objective: Include both generation costs and storage costs
    generation_cost = pulp.lpSum(
        gen_time_series.loc[
            (gen_time_series['id'] == g) & (gen_time_series['time'] == t),
            'gencost'
        ].values[0] * GEN[g, t]
        for g in G for t in T
    )

    # Simple storage cost to prevent unnecessary cycling
    storage_cost = pulp.lpSum(
        0.001 * (P_discharge[s, t] + P_charge[s, t])
        for s in S for t in T
    )

    DCOPF += generation_cost + storage_cost, "Total_Cost"

    # Get load buses from bus dataframe
    load_buses = bus[bus['type'] == 1]['bus_i'].values

    # 8. Power Balance Constraints
    for t in T:
        for i in N:  # Include all buses, even those without generators
            # sum non-storage gen at bus i
            gen_sum = pulp.lpSum(
                GEN[g, t]
                for g in G
                if gen_time_series.loc[
                    (gen_time_series['id'] == g) & (gen_time_series['time'] == t),
                    'bus'
                ].values[0] == i
            )

            # Get demand at bus i - only if it's a load bus
            pd_val = 0
            if i in load_buses:
                demands_at_bus = demand_time_series.loc[
                    (demand_time_series['bus'] == i) & (demand_time_series['time'] == t),
                    'pd'
                ]
                pd_val = demands_at_bus.sum() if not demands_at_bus.empty else 0

            # Storage at bus i => discharge - charge
            storages_at_bus_i = gen_time_series.loc[
                (gen_time_series['bus'] == i) & (gen_time_series['emax'] > 0),
                'id'
            ].unique()

            if len(storages_at_bus_i) > 0:
                gen_sum += pulp.lpSum(
                    (P_discharge[s, t] - P_charge[s, t]) for s in storages_at_bus_i
                )

            # Power flow balance at each bus
            flow_out = pulp.lpSum(FLOW[i, j, t] for j in branch.loc[branch['fbus'] == i, 'tbus'])
```

```python
            flow_in  = pulp.lpSum(FLOW[j, i, t] for j in branch.loc[branch['
                ↪ tbus'] == i, 'fbus'])

            DCOPF += (gen_sum - pd_val + flow_in - flow_out == 0), f"
                ↪ Power_Balance_Bus_{i}_Time_{t}"

    # 9. Flow limits
    for _, row_b in branch.iterrows():
        i = row_b['fbus']
        j = row_b['tbus']
        rate_a = row_b['ratea']
        for t in T:
            DCOPF += FLOW[i, j, t] <= rate_a,  f"Flow_Limit_{i}_{j}
                ↪ _Upper_Time_{t}"
            DCOPF += FLOW[i, j, t] >= -rate_a, f"Flow_Limit_{i}_{j}
                ↪ _Lower_Time_{t}"

    # 10. Solve
    print("[DCOPF] About to solve the LP problem with CBC solver...")
    solver_result = DCOPF.solve(pulp.PULP_CBC_CMD(msg=True))

    status_code = DCOPF.status
    status_str = pulp.LpStatus[status_code]
    print(f"[DCOPF] Solver returned status code = {status_code}, interpreted
        ↪  as '{status_str}'")

    # If code != 1 => Not recognized as Optimal
    if status_code != 1:
        print(f"[DCOPF] Not optimal => returning None.")
        return None

    # 11. Extract results
    print("[DCOPF] Extraction of results - building final dictionary...")

    # a) Non-storage generation
    generation = []
    for g in G:
        g_bus = gen_time_series.loc[gen_time_series['id'] == g, 'bus'].iloc
            ↪ [0]
        for t in T:
            val = pulp.value(GEN[g, t])
            generation.append({
                'time': t,
                'id': g,
                'node': g_bus,
                'gen': 0 if math.isnan(val) else val
            })
    generation = pd.DataFrame(generation)

    # b) Storage net output
    storage_generation = []
    for s in S:
        s_bus = gen_time_series.loc[gen_time_series['id'] == s, 'bus'].iloc
            ↪ [0]
        for t in T:
            ch = pulp.value(P_charge[s, t])
            dis = pulp.value(P_discharge[s, t])
            if math.isnan(ch):
                ch = 0
            if math.isnan(dis):
                dis = 0
            net_out = dis - ch
            storage_generation.append({
```

```
                'time': t,
                'id': s,
                'node': s_bus,
                'gen': net_out
            })
    storage_generation = pd.DataFrame(storage_generation)
    generation = pd.concat([generation, storage_generation], ignore_index=True)

    # c) Angles
    angles = []
    for i_bus in N:
        for t in T:
            val_theta = pulp.value(THETA[i_bus, t])
            angles.append({
                'time': t,
                'bus': i_bus,
                'theta': 0 if math.isnan(val_theta) else val_theta
            })
    angles = pd.DataFrame(angles)

    # d) Flows
    flows_list = []
    for (i_bus, j_bus, t) in FLOW:
        val_flow = pulp.value(FLOW[i_bus, j_bus, t])
        flows_list.append({
            'time': t,
            'from_bus': i_bus,
            'to_bus': j_bus,
            'flow': 0 if math.isnan(val_flow) else val_flow
        })
    flows_df = pd.DataFrame(flows_list)

    # e) Storage states
    storage_list = []
    for s in S:
        for idx_t, tt in enumerate(extended_T):
            E_val = pulp.value(E[s, tt])
            Pch = pulp.value(P_charge[s, tt]) if tt in T else None
            Pdis = pulp.value(P_discharge[s, tt]) if tt in T else None
            storage_list.append({
                'storage_id': s,
                'time': tt,
                'E': 0 if math.isnan(E_val) else E_val,
                'P_charge': None if (Pch is None or math.isnan(Pch)) else Pch,
                'P_discharge': None if (Pdis is None or math.isnan(Pdis)) else Pdis
            })

    # Always define columns so groupby('storage_id') won't fail
    storage_df = pd.DataFrame(
        storage_list,
        columns=["storage_id", "time", "E", "P_charge", "P_discharge"]
    )

    if len(S) > 0 and not storage_df.empty:
        # Shift E if you want SoC at start of each interval
        storage_corrected = []
        for s_id, group in storage_df.groupby('storage_id'):
            group = group.sort_values('time').reset_index(drop=True)
            group['E'] = group['E'].shift(-1)
            # remove last row
```

```python
            group = group.iloc[:-1]
            storage_corrected.append(group)
        storage_df = pd.concat(storage_corrected, ignore_index=True)

    total_cost = pulp.value(DCOPF.objective)
    if total_cost is None:
        print("[DCOPF] Warning: Could not extract objective value. Setting
            ↪ cost to infinity.")
        total_cost = float('inf')

    status = pulp.LpStatus[DCOPF.status]

    print(f"[DCOPF] Final cost = {total_cost}, status = {status}")
    print("[DCOPF] Done, returning result dictionary.")

    return {
        'generation': generation,
        'angles': angles,
        'flows': flows_df,
        'storage': storage_df,
        'cost': total_cost,
        'status': status
    }
```

Listing 1: DCOPF Implementation

A.2 Solver

```python
#!/usr/bin/env python3

"""
multi_scenario.py

- Loads scenarios from scenarios_parameters.csv
- Runs DCOPF for each scenario across winter, summer, autumn_spring
- Saves results and plots in /data/results/<scenario_name>/
- Summarizes costs in scenario_results.csv
"""

import os
import sys

# Add the scripts directory to Python path
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

import pandas as pd
import numpy as np
import ast

from dcopf import dcopf
from dotenv import load_dotenv
from scenario_critic import ScenarioCritic
from update_readme import update_readme_with_scenarios,
    ↪ create_readme_template, get_project_root
from create_master_invest import InvestmentAnalysis
from visualization.summary_plots import create_annual_summary_plots,
    ↪ create_scenario_comparison_plot
from visualization.scenario_plots import plot_scenario_results
from core.time_series import build_gen_time_series, build_demand_time_series
from core.helpers import ask_user_confirmation
```

```python
from typing import Dict, Any, List, Optional
from dataclasses import dataclass
from visualization.report_plots import create_scenario_plots

# Paths
project_root = get_project_root()
working_dir = os.path.join(project_root, "data", "working")
results_root = os.path.join(project_root, "data", "results")

bus_file = os.path.join(working_dir, "bus.csv")
branch_file = os.path.join(working_dir, "branch.csv")
master_gen_file = os.path.join(working_dir, "master_gen.csv")
master_load_file = os.path.join(working_dir, "master_load.csv")
scenarios_params_file = os.path.join(working_dir, "scenarios_parameters.csv")

# Season weights
season_weights = {
    "winter": 13,
    "summer": 13,
    "autumn_spring": 26
}

# Load environment variables
load_dotenv('../.env.local')
api_key = os.getenv('OPENAPI_KEY')
if not api_key:
    raise ValueError("OpenAI API key not found in .env.local file")

# Initialize critic
critic = ScenarioCritic(api_key)

# Data classes
@dataclass
class SeasonalData:
    generation: Dict[str, float]  # Asset type -> generation amount
    cost: float
    capacity_factors: Dict[str, float]

@dataclass
class ScenarioVariant:
    scenario_name: str
    variant_type: str  # 'nominal', 'high', 'low'
    load_factor: float
    annual_cost: float
    seasonal_data: Dict[str, SeasonalData]  # season -> data
    generation_by_asset: Dict[str, float]
    generation_costs: Dict[str, float]
    available_capacity: Dict[str, float]
    capacity_factors: Dict[str, float]

    @property
    def full_name(self) -> str:
        return f"{self.scenario_name}_{self.variant_type}"

    def to_dict(self) -> Dict: # Convert to flat dictionary for DataFrame
        """Convert to flat dictionary for DataFrame"""
        result = {
            "scenario_name": self.full_name,
            "base_scenario": self.scenario_name,
            "variant": self.variant_type,
            "load_factor": self.load_factor,
            "annual_cost": self.annual_cost
```

```
        }

        # Add seasonal data
        for season, data in self.seasonal_data.items():
            for asset, gen in data.generation.items():
                result[f"{season}_gen_{asset}"] = gen
            result[f"{season}_cost"] = data.cost

        # Add annual metrics
        for asset, gen in self.generation_by_asset.items():
            result[f"gen_{asset}"] = gen
            result[f"gen_cost_{asset}"] = self.generation_costs.get(asset, 0)
            result[f"avail_gen_{asset}"] = self.available_capacity.get(asset, 0)
            result[f"capacity_factor_{asset}"] = self.capacity_factors.get(asset, 0)

        return result

# Run a single scenario variant (nominal, high, or low load)
def run_scenario_variant(
    scenario_name: str,
    gen_positions: Dict[int, int],
    storage_positions: Dict[int, int],
    load_factor: float,
    variant: str,
    data_context: Dict[str, Any]
) -> Optional[ScenarioVariant]:
    """Run a single scenario variant (nominal, high, or low load)"""

    seasonal_data = {}
    total_gen_year = {}
    total_gen_cost_year = {}
    total_avail_gen_year = {}

    print(f"\nProcessing {scenario_name} ({variant} load) with:")
    print(f"  Load factor: {load_factor}")

    for season in ["winter", "summer", "autumn_spring"]:
        print(f"  Running {season}...")
        season_result = run_single_season(
            season=season,
            gen_positions=gen_positions,
            storage_positions=storage_positions,
            load_factor=load_factor,
            data_context=data_context
        )

        if season_result is None:
            return None

        # Convert SeasonResult to SeasonalData
        generation_dict = {
            asset: metrics.generation
            for asset, metrics in season_result.metrics.items()
        }
        capacity_factors = {
            asset: (metrics.generation / metrics.available if metrics.available > 0 else 0)
            for asset, metrics in season_result.metrics.items()
        }

```

```python
        seasonal_data[season] = SeasonalData(
            generation=generation_dict,
            cost=season_result.cost,
            capacity_factors=capacity_factors
        )

        # Accumulate annual metrics
        weight = data_context['season_weights'][season]
        for asset, metrics in season_result.metrics.items():
            total_gen_year[asset] = total_gen_year.get(asset, 0) + metrics.generation * weight
            total_gen_cost_year[asset] = total_gen_cost_year.get(asset, 0) + metrics.cost * weight
            total_avail_gen_year[asset] = total_avail_gen_year.get(asset, 0) + metrics.available * weight

    # Calculate capacity factors
    capacity_factors = {
        asset: total_gen_year[asset] / total_avail_gen_year[asset]
        if total_avail_gen_year.get(asset, 0) > 0 else 0
        for asset in total_gen_year
    }

    annual_cost = sum(
        data.cost * data_context['season_weights'][season]
        for season, data in seasonal_data.items()
    )

    return ScenarioVariant(
        scenario_name=scenario_name,
        variant_type=variant,
        load_factor=load_factor,
        annual_cost=annual_cost,
        seasonal_data=seasonal_data,
        generation_by_asset=total_gen_year,
        generation_costs=total_gen_cost_year,
        available_capacity=total_avail_gen_year,
        capacity_factors=capacity_factors
    )


def load_data_context() -> Dict[str, Any]:
    """
    Load and preprocess all required data for scenario analysis.
    Returns a context dictionary containing all necessary data and mappings.
    """
    # Load base data files
    bus = pd.read_csv(bus_file)
    branch = pd.read_csv(branch_file)
    master_gen = pd.read_csv(master_gen_file, parse_dates=["time"]).sort_values("time")
    master_load = pd.read_csv(master_load_file, parse_dates=["time"]).sort_values("time")
    scenarios_df = pd.read_csv(scenarios_params_file)

    # Process branch data
    branch.rename(columns={"rateA": "ratea"}, inplace=True, errors="ignore")
    branch["sus"] = 1 / branch["x"]
    branch["id"] = np.arange(1, len(branch) + 1)

    # Create mappings
    id_to_type = master_gen.drop_duplicates(subset=['id'])[['id', 'type']].set_index('id')['type'].to_dict()
```

```python
    type_to_id = master_gen.drop_duplicates(subset=['type'])[['type', 'id']].set_index('type')['id'].to_dict()
    id_to_gencost = master_gen.drop_duplicates(subset=['id'])[['id', 'gencost']].set_index('id')['gencost'].to_dict()
    id_to_pmax = master_gen.drop_duplicates(subset=['id'])[['id', 'pmax']].set_index('id')['pmax'].to_dict()

    return {
        # Raw data
        'bus': bus,
        'branch': branch,
        'master_gen': master_gen,
        'master_load': master_load,
        'scenarios_df': scenarios_df,

        # Mappings
        'id_to_type': id_to_type,
        'type_to_id': type_to_id,
        'id_to_gencost': id_to_gencost,
        'id_to_pmax': id_to_pmax,

        # Constants
        'season_weights': season_weights,

        # Paths
        'results_root': results_root
    }

def parse_positions(positions_str: str, type_to_id: Dict[str, int]) -> Dict[int, int]:
    """
    Parse positions string from scenarios file and convert types to IDs.

    Args:
        positions_str: String representation of positions dictionary
        type_to_id: Mapping from generator type to ID

    Returns:
        Dictionary mapping bus numbers to generator IDs
    """
    try:
        positions_raw = ast.literal_eval(positions_str)
        return {
            int(bus): type_to_id[gen_type]
            for bus, gen_type in positions_raw.items()
        }
    except (ValueError, KeyError) as e:
        print(f"Error parsing positions: {e}")
        return {}

@dataclass
class SeasonMetrics:
    generation: float
    cost: float
    available: float

@dataclass
class SeasonResult:
    metrics: Dict[str, SeasonMetrics]
    cost: float
    storage_data: Optional[pd.DataFrame] = None

def run_single_season(
```

```
    season: str,
    gen_positions: Dict[int, int],
    storage_positions: Dict[int, int],
    load_factor: float,
    data_context: Dict[str, Any]
) -> Optional[SeasonResult]:
    """Run DCOPF for a single season and collect results"""
    # Build time series
    gen_ts = build_gen_time_series(
        data_context['master_gen'],
        gen_positions,
        storage_positions,
        season
    )

    # Build demand time series
    demand_ts = build_demand_time_series(
        data_context['master_load'],
        load_factor,
        season
    )

    # Print debug info
    print(f"\nAssets in {season}:")
    print("Generators:", gen_positions)
    print("Storage:", storage_positions)
    print("Types in time series:", gen_ts['type'].unique())
    print(f"Load factor: {load_factor}")

    # Run DCOPF
    results = dcopf(
        gen_ts,
        data_context['branch'],
        data_context['bus'],
        demand_ts,
        delta_t=1
    )

    # Debug prints to check DCOPF results structure
    print("\nDCOPF Results Structure:")
    print("Available keys:", results.keys())
    if 'storage' in results:
        print("\nStorage data found!")
        print("Storage data columns:", results['storage'].columns.tolist())
        print("First few rows of storage data:")
        print(results['storage'].head())
        print("\nStorage data shape:", results['storage'].shape)
        print("Storage data types:", results['storage'].dtypes)
    else:
        print("\nNo storage data in DCOPF results")

    if not results or results.get("status") != "Optimal":
        print(f"Failed to find optimal solution for {season}")
        return None

    # Process generation metrics
    metrics_by_type = {}

    # Group generation by type
    for _, gen_row in results['generation'].iterrows():
        gen_type = data_context['id_to_type'].get(gen_row['id'])
        if gen_type:
            if gen_type not in metrics_by_type:
```

```python
                metrics_by_type[gen_type] = SeasonMetrics(
                    generation=0,
                    cost=0,
                    available=0
                )

            # Add generation
            metrics_by_type[gen_type].generation += gen_row['gen']

            # Add cost
            if gen_row['id'] in data_context['id_to_gencost']:
                cost = gen_row['gen'] * data_context['id_to_gencost'][gen_row['id']]
                metrics_by_type[gen_type].cost += cost

            # Calculate available capacity
            if gen_row['id'] in data_context['id_to_pmax']:
                metrics_by_type[gen_type].available += data_context['id_to_pmax'][gen_row['id']]

    # Extract storage data if available
    storage_data = None
    if 'storage' in results:
        storage_data = results['storage'].copy()
        # Use 'E' column as Storage_SoC (as in your PULP code)
        if 'E' in storage_data.columns:
            storage_data['Storage_SoC'] = storage_data['E']
            storage_data.set_index('time', inplace=True)

    return SeasonResult(
        metrics=metrics_by_type,
        cost=results.get("cost", 0.0),
        storage_data=storage_data
    )

class MultiScenario:
    """Main class to handle multiple scenario analysis for power system investments"""

    #######################
    # 1. INITIALIZATION
    #######################
    def __init__(self, plot_gen_mix=False):
        """Initialize parameters, paths, and configurations"""
        # Setup paths
        self.setup_paths()

        # Load scenario parameters
        self.load_scenario_parameters()

        # Initialize analysis parameters
        self.init_analysis_parameters()

        # Setup output directories
        self.create_output_dirs()

    #######################
    # 2. NETWORK CREATION
    #######################
    def create_network(self):
        """Create and configure PyPSA network for scenarios"""
        # Load component data (generators, buses, branches)
        self.load_component_data()
```

```python

        # Configure network parameters
        self.setup_network_parameters()

        # Add components to network
        self.add_network_components()

        return network

    ######################
    # 3. SCENARIO SOLVING
    ######################
    def solve(self):
        """Main solving function for all scenarios"""
        # Iterate through scenarios
        for scenario in self.scenarios:
            # Create network for scenario
            network = self.create_network()

            # Run OPF
            self.run_opf(network)

            # Store results
            self.store_scenario_results(network)

            # Calculate metrics
            self.calculate_scenario_metrics()

    ######################
    # 4. METRICS CALCULATION
    ######################
    def calculate_metrics(self):
        """Calculate investment and performance metrics"""
        # Financial calculations
        self.calculate_financial_metrics()

        # Sensitivity analysis
        self.perform_sensitivity_analysis()

        # Store metric results
        self.store_metrics()

    ######################
    # 5. VISUALIZATION
    ######################
    def generate_plots(self):
        """Generate all required plots"""
        # Generation mix plots
        self.plot_generation_mix()

        # Investment metric plots
        self.plot_investment_metrics()

        # Sensitivity analysis plots
        self.plot_sensitivity_results()

        # Add AI comments to plots
        self.add_plot_comments()

    ######################
    # 6. REPORTING
    ######################
    def create_summary(self):
```

```python
        """Create summary reports and analysis"""
        # Generate summary statistics
        self.calculate_summary_stats()

        # Create summary file
        self.write_summary_file()

        # Generate AI analysis
        self.generate_ai_analysis()

    #######################
    # 7. UTILITY FUNCTIONS
    #######################
    def setup_paths(self):
        """Setup directory paths"""
        pass

    def load_scenario_parameters(self):
        """Load and validate scenario parameters"""
        pass

    def store_scenario_results(self, network):
        """Store results for a specific scenario"""
        pass

def main():
    # Load all data
    data_context = load_data_context()

    # Ask for sensitivity analysis
    run_sensitivity = ask_user_confirmation(
        "Do you want to run sensitivity analysis?"
    )

    scenario_variants: List[ScenarioVariant] = []

    # Dictionary to collect storage data from all scenarios
    all_scenarios_storage = {}

    for _, row in data_context['scenarios_df'].iterrows():
        scenario_name = row["scenario_name"]
        gen_positions = parse_positions(row["gen_positions"], data_context['type_to_id'])
        storage_positions = parse_positions(row["storage_units"], data_context['type_to_id'])
        base_load_factor = float(row["load_factor"])

        # Storage data collection for nominal load only
        storage_data = pd.DataFrame()

        # Run variants
        variants_to_run = [("nominal", base_load_factor)]
        if run_sensitivity:
            variants_to_run.extend([
                ("high", base_load_factor * 1.2),
                ("low", base_load_factor * 0.8)
            ])

        for variant_name, load_factor in variants_to_run:
            # Run scenario and collect results
            for season in ["winter", "summer", "autumn_spring"]:
                season_result = run_single_season(
                    season=season,
```

```python
                    gen_positions=gen_positions,
                    storage_positions=storage_positions,
                    load_factor=load_factor,
                    data_context=data_context
                )

                # Collect storage data for nominal load case
                if variant_name == "nominal" and season_result and season_result.storage_data is not None:
                    storage_data = pd.concat([storage_data, season_result.storage_data])

            result = run_scenario_variant(
                scenario_name=scenario_name,
                gen_positions=gen_positions,
                storage_positions=storage_positions,
                load_factor=load_factor,
                variant=variant_name,
                data_context=data_context
            )
            if result:
                scenario_variants.append(result)

        # Store storage data if available
        if not storage_data.empty:
            storage_data = storage_data.sort_index()
            all_scenarios_storage[scenario_name] = storage_data

            # Create plots with this scenario's data
            if 'Storage_SoC' in storage_data.columns:
                create_scenario_plots({scenario_name: storage_data})
                print(f"Created storage plots for scenario {scenario_name}")
            else:
                print(f"Warning: No Storage_SoC column found in scenario {scenario_name}")

    # Convert to DataFrame
    results_df = pd.DataFrame([
        variant.to_dict() for variant in scenario_variants
    ])

    # Save initial results
    results_df.to_csv(os.path.join(results_root, "scenario_results.csv"), index=False)
    print("Initial results saved to CSV.")

    # Then perform investment analysis
    print("\nPerforming investment analysis...")
    analysis = InvestmentAnalysis()
    investment_results = analysis.analyze_scenario(
        os.path.join(results_root, "scenario_results.csv"),
        master_gen_file
    )

    print("Investment analysis columns:", investment_results.columns.tolist())

    # Check if base_scenario is in the index
    if 'base_scenario' in investment_results.index.names:
        # Reset index only if base_scenario is not already a column
        if 'base_scenario' not in investment_results.columns:
            investment_results = investment_results.reset_index()

```

```python
    # Filter for nominal variants
    nominal_results = results_df[results_df['variant'] == 'nominal'].copy()

    # Get the actual columns that exist in the DataFrame
    available_columns = nominal_results.columns.tolist()
    print("\nAvailable columns in nominal_results:", available_columns)

    # Define essential columns based on what's available
    base_essential_columns = [
        'base_scenario',
        'variant',
        'load_factor',
        'annual_cost'
    ]

    # Add generation columns that exist
    gen_columns = [col for col in available_columns if col.startswith('gen_'
        ↪ )]
    essential_columns = base_essential_columns + gen_columns

    print("\nSelected essential columns:", essential_columns)

    # Filter columns
    nominal_results = nominal_results[essential_columns]

    # Merge the results
    final_results = nominal_results.merge(
        investment_results,
        on='base_scenario',
        how='left'
    )

    # Add a clean scenario identifier
    final_results['scenario_id'] = final_results.apply(
        lambda x: f"{x['base_scenario']}_{x['variant']}", axis=1
    )

    # Define base columns that we want first
    base_columns = [
        'scenario_id',
        'base_scenario',
        'variant',
        'load_factor'
    ]

    # Define investment-related columns
    investment_columns = [
        'installed_capacity',
        'initial_investment',
        'annual_cost',
        'annual_costs'
    ]

    # Define NPV and annuity columns
    financial_columns = [
        'npv_10y', 'npv_20y', 'npv_30y',
        'annuity_10y', 'annuity_20y', 'annuity_30y'
    ]

    # Get generation-related columns
    generation_columns = [col for col in final_results.columns
                          if col.startswith('gen_') or
                             col.startswith('winter_') or
```

```python
                            col.startswith('summer_') or
                            col.startswith('autumn_')]

    # Combine all columns in desired order
    column_order = (base_columns +
                    investment_columns +
                    financial_columns +
                    generation_columns)

    # Add any remaining columns that we haven't explicitly ordered
    remaining_columns = [col for col in final_results.columns
                        if col not in column_order]
    column_order.extend(remaining_columns)

    # Reorder columns, but only include ones that exist
    final_columns = [col for col in column_order
                     if col in final_results.columns]
    final_results = final_results[final_columns]

    # Save the final results
    final_results.to_csv(os.path.join(results_root, "scenario_results_with_investment.csv"),
                         index=False)

    # Ask user for generation preferences
    generate_plots = ask_user_confirmation("Do you want to generate plots?")
    generate_individual = ask_user_confirmation("Do you want to generate individual scenario reports?")
    generate_global = ask_user_confirmation("Do you want to generate a global comparison report?")

    if generate_plots:
        print("\nGenerating plots...")
        # Group scenarios by base scenario
        scenario_groups = final_results.groupby('base_scenario')

        for base_scenario, group in scenario_groups:
            print(f"\nProcessing scenario: {base_scenario}")

            # Get variants with debug printing
            nominal_data = group[group['variant'] == 'nominal'].iloc[0].to_dict()

            # Get high variant
            high_data = {}
            high_variant = group[group['variant'] == 'high']
            if not high_variant.empty:
                high_data = high_variant.iloc[0].to_dict()
                print(f"Found high variant for {base_scenario}")
            else:
                print(f"No high variant for {base_scenario}")

            # Get low variant
            low_data = {}
            low_variant = group[group['variant'] == 'low']
            if not low_variant.empty:
                low_data = low_variant.iloc[0].to_dict()
                print(f"Found low variant for {base_scenario}")
            else:
                print(f"No low variant for {base_scenario}")

            # Add sensitivity data to nominal data
            nominal_data['high_variant'] = high_data
```

```
            nominal_data['low_variant'] = low_data

            # Create plots
            create_annual_summary_plots(nominal_data, results_root)

    if generate_individual or generate_global:
        print("\nGenerating requested reports...")

        # Generate individual reports if requested
        if generate_individual:
            print("\nGenerating individual scenario reports...")
            # Only process nominal variants for reports
            nominal_results = final_results[final_results['variant'] == 'nominal']
            for _, row in nominal_results.iterrows():
                if row['annual_cost'] is not None:
                    critic.analyze_scenario(row.to_dict(), results_root)
            print("Individual reports completed.")

        # Generate global report if requested
        if generate_global:
            print("\nGenerating global comparison report...")
            # Use only nominal variants for global comparison
            nominal_results = final_results[final_results['variant'] == 'nominal']
            critic.create_global_comparison_report(nominal_results, results_root)
            print("Global report completed.")

        print("All requested reports generated.")
    else:
        print("\nSkipping report generation.")

    # Update README with scenario links
    project_root = get_project_root()
    readme_path = os.path.join(project_root, 'README.md')
    create_readme_template(readme_path)  # Create/update the full README
    update_readme_with_scenarios()       # Update the scenario links

if __name__ == "__main__":
    main()
```

Listing 2: Solver

A.3 Economic Calculations

```
import pandas as pd
import numpy as np
import os

def get_project_root():
    """Get the absolute path to the project root directory"""
    current_dir = os.path.dirname(os.path.abspath(__file__))  # scripts directory
    project_root = os.path.dirname(current_dir)  # up one level to project root
    return project_root

class InvestmentAnalysis:
    def __init__(self):
```

```
        """Initialize with updated CAPEX values"""
        # Investment costs (CAPEX) per MW
        self.capex = {
            'wind': 1400000,      # CHF/MW
            'solar': 900000,      # CHF/MW
            'battery1': 250000,   # CHF/MW
            'battery2': 450000,   # CHF/MWy
        }

        # Financial parameters
        self.discount_rate = 0.08   # 8%
        self.time_horizons = [10, 20, 30]  # Years to calculate NPV for

        # Technical lifetime of assets
        self.lifetime = {
            'wind': 19,
            'solar': 25,
            'battery1': 6,
            'battery2': 8,
        }

        self.annual_opex_percent = {
            'wind': 0.04,
            'solar': 0.02,
            'battery1': 0.03,
            'battery2': 0.04,
        }

        # Add load factors for variants
        self.load_factors = {
            'low': 0.8,
            'nominal': 1.0,
            'high': 1.2
        }

    def calculate_initial_investment(self, installed_capacity):
        """Calculate initial investment based on installed capacities"""
        investment = 0
        for tech, capacity in installed_capacity.items():
            # Convert technology name to match capex keys if needed
            tech_key = tech.lower()  # Convert to lowercase to match capex
                ↪ keys
            if tech_key in self.capex:
                investment += capacity * self.capex[tech_key]
                print(f"Adding investment for {tech_key}: {capacity} MW * ${
                    ↪ self.capex[tech_key]}/MW = ${capacity * self.capex[
                    ↪ tech_key]}")

        print(f"Total initial investment: ${investment}")
        return investment

    def calculate_annual_costs(self, operational_costs, installed_capacity):
        """Calculate total annual costs including O&M"""
        annual_cost = operational_costs  # From scenario results

        # Add maintenance costs
        for tech, capacity in installed_capacity.items():
            if tech in self.annual_opex_percent:
                maintenance = capacity * self.capex[tech] * self.
                    ↪ annual_opex_percent[tech]
                annual_cost += maintenance

        return annual_cost
```

```

    def calculate_npv(self, initial_investment, annual_costs, years, installed_capacity):
        """Calculate NPV for a specific time horizon"""
        npv = -initial_investment
        for year in range(years):
            # Add replacement costs if asset lifetime is exceeded
            replacement_cost = 0
            if year > 0:  # Check for replacements
                for tech, lifetime in self.lifetime.items():
                    if year % lifetime == 0:  # Time to replace
                        capacity = installed_capacity.get(tech, 0)
                        replacement_cost += capacity * self.capex[tech]

            yearly_cashflow = -annual_costs - replacement_cost
            npv += yearly_cashflow / (1 + self.discount_rate)**(year + 1)
        return npv

    def calculate_annuity(self, npv, years):
        """Calculate annuity payment for a specific time horizon"""
        if npv >= 0:  # For positive NPV, return 0 (no payments needed)
            return 0
        annuity_factor = (self.discount_rate * (1 + self.discount_rate)**years) / \
                        ((1 + self.discount_rate)**years - 1)
        return -npv * annuity_factor  # Negative NPV becomes positive annuity

    def analyze_scenario(self, scenario_results_path, master_gen_path):
        """Main analysis function"""
        try:
            # Get project root for path resolution
            project_root = get_project_root()

            # Resolve absolute paths
            scenario_results_path = os.path.join(project_root, 'data', 'results', 'scenario_results.csv')
            working_dir = os.path.join(project_root, 'data', 'working')
            scenarios_params_path = os.path.join(working_dir, 'scenarios_parameters.csv')

            # Load data
            scenario_results = pd.read_csv(scenario_results_path)
            scenarios_params = pd.read_csv(scenarios_params_path)

            results_list = []  # Change to list to store multiple variants
            for scenario in scenario_results['base_scenario'].unique():
                print(f"\n{'='*50}")
                print(f"Processing scenario: {scenario}")

                # Get scenario configuration
                scenario_config = scenarios_params[scenarios_params['scenario_name'] == scenario].iloc[0]
                gen_positions = eval(scenario_config['gen_positions'])
                storage_positions = eval(scenario_config['storage_units'])

                # Count installed capacity
                installed_capacity = {}
                for _, gen_type in gen_positions.items():
                    tech_key = gen_type.lower()
                    installed_capacity[tech_key] = installed_capacity.get(tech_key, 0) + 1

```

```
            for _, storage_type in storage_positions.items():
                installed_capacity[storage_type] = installed_capacity.get(storage_type, 0) + 1

            # Calculate base values
            initial_inv = self.calculate_initial_investment(installed_capacity)

            # Process each variant
            for variant, load_factor in self.load_factors.items():
                scenario_id = f"{scenario}_{variant}"
                print(f"\nProcessing variant: {variant} (load factor: {load_factor})")

                # Get scenario data for the variant
                scenario_data = scenario_results[
                    (scenario_results['base_scenario'] == scenario) &
                    (scenario_results['variant'] == 'nominal')
                ].iloc[0]

                # Scale costs based on load factor
                base_annual_cost = float(scenario_data['annual_cost'])
                scaled_annual_cost = base_annual_cost * load_factor
                annual_costs = self.calculate_annual_costs(scaled_annual_cost, installed_capacity)

                # Calculate NPV and annuity for different time horizons
                npv_results = {}
                annuity_results = {}
                for years in self.time_horizons:
                    npv = self.calculate_npv(initial_inv, annual_costs, years, installed_capacity)
                    annuity = self.calculate_annuity(npv, years)
                    npv_results[f'npv_{years}y'] = npv
                    annuity_results[f'annuity_{years}y'] = annuity

                # Scale generation values
                gen_results = {}
                for col in scenario_data.index:
                    if col.startswith('gen_'):
                        base_value = float(scenario_data[col])
                        gen_results[col] = base_value * load_factor

                # Compile results for this variant
                variant_results = {
                    'scenario_id': scenario_id,
                    'base_scenario': scenario,
                    'variant': variant,
                    'load_factor': load_factor,
                    'installed_capacity': str(installed_capacity),
                    'initial_investment': initial_inv,
                    'annual_cost': scaled_annual_cost,
                    'annual_costs': annual_costs,
                    **npv_results,
                    **annuity_results,
                    **gen_results,
                    'scenario_name': scenario
                }

                results_list.append(variant_results)

        # Convert results to DataFrame
        results_df = pd.DataFrame(results_list)
```

```
            # Save results
            output_path = os.path.join(project_root, 'data', 'results', 'scenario_results_with_investment.csv')
            results_df.to_csv(output_path, index=False)
            print(f"\nResults saved to {output_path}")

            return results_df

        except Exception as e:
            print(f"Error in analyze_scenario: {str(e)}")
            raise

# If running this file directly
if __name__ == '__main__':
    project_root = get_project_root()
    analysis = InvestmentAnalysis()

    # Use proper paths relative to project root
    results = analysis.analyze_scenario(
        os.path.join(project_root, 'data', 'results', 'scenario_results.csv'),
        os.path.join(project_root, 'data', 'working', 'master_gen.csv')
    )

    print(results)

    # Format results for display
    display_df = results.copy()

    # Format monetary values
    monetary_columns = ['initial_investment', 'annual_costs'] + \
                      [f'npv_{y}y' for y in analysis.time_horizons] + \
                      [f'annuity_{y}y' for y in analysis.time_horizons]

    for col in monetary_columns:
        display_df[col] = display_df[col].map('${:,.2f}'.format)

    # Format installed capacity
    display_df['installed_capacity'] = display_df['installed_capacity'].apply(
        lambda x: '\n'.join([f"{k}: {v:.2f} MW" for k, v in x.items()])
    )

    # Sort by 30-year NPV and get top 10
    top_10 = display_df.sort_values('npv_30y', ascending=True).head(10)

    print("\n=== Top 10 Scenarios by 30-year NPV ===")
    print("\nDetailed Results:")
    for idx, row in top_10.iterrows():
        print(f"\nScenario: {idx}")
        print("Installed Capacity:")
        print(row['installed_capacity'])
        print(f"Initial Investment: {row['initial_investment']}")
        print(f"Annual Costs: {row['annual_costs']}")
        print("\nNPV Analysis:")
        for years in analysis.time_horizons:
            print(f"{years}-year NPV: {row[f'npv_{years}y']}")
            print(f"{years}-year Annuity: {row[f'annuity_{years}y']}")
        print("-" * 50)

    # Alternative: Create an Excel file with formatted results
    writer = pd.ExcelWriter('data/results/investment_analysis.xlsx', engine=
```

```
'xlsxwriter')

    # Write to Excel with formatting
    display_df.to_excel(writer, sheet_name='All Results')
    top_10.to_excel(writer, sheet_name='Top 10 Scenarios')

    # Get workbook and worksheet objects
    workbook = writer.book
    worksheet = writer.sheets['All Results']

    # Add formatting
    money_format = workbook.add_format({'num_format': '$#,##0.00'})
    wrap_format = workbook.add_format({'text_wrap': True})

    # Set column widths
    worksheet.set_column('B:B', 40, wrap_format)  # Installed capacity
    worksheet.set_column('C:F', 15, money_format)  # Money columns

    writer.close()

    print("\nResults have been saved to 'data/results/investment_analysis.xlsx'")

    # Get raw results sorted by NPV
    sorted_results = results.sort_values('npv', ascending=True)

    # Get specific metrics
    npv_series = sorted_results['npv']
    annuities = sorted_results['annuity']
```

A.4 AI Integration and Reporting

```
from openai import OpenAI
import pandas as pd
from typing import Dict, Any
from datetime import datetime
import os
import matplotlib.pyplot as plt
import re

class ScenarioCritic:
    def __init__(self, api_key: str):
        """Initialize the critic with OpenAI API key"""
        self.client = OpenAI(api_key=api_key)

        self.context_prompt = """
        You are analyzing energy system scenarios with different mixes of generation sources.
        The analysis includes:
        - Annual operational costs
        - Generation per asset type
        - Generation costs per asset type
        - Capacity factors
        - NPVs and annuity

        Technologies involved may include:
        - Nuclear
        - Gas
        - Wind
        - Solar
```

```python
        - Battery storage systems

        The goal is to evaluate the economic efficiency and technical feasibility of different energy mix scenarios.
        Output in markdown format.
        """

    def generate_critique(self, scenario_data: Dict[str, Any]) -> str:
        """Generate a critique for a single scenario using OpenAI API"""

        # Format the generation and cost data
        gen_data = {k: v for k, v in scenario_data.items() if k.startswith('gen_')}
        cost_data = {k: v for k, v in scenario_data.items() if k.startswith('gen_cost_')}
        capacity_factors = {k: v for k, v in scenario_data.items() if k.startswith('capacity_factor_')}

        # Create formatted strings for each section
        gen_lines = '\n'.join([f'- {k.replace("gen_", "")}: {v} MW' for k, v in gen_data.items()])
        cost_lines = '\n'.join([f'- {k.replace("gen_cost_", "")}: {v}' for k, v in cost_data.items()])
        cf_lines = '\n'.join([f'- {k.replace("capacity_factor_", "")}: {v}' for k, v in capacity_factors.items()])

        scenario_prompt = f"""Scenario Analysis Results:

Scenario Name: {scenario_data.get('scenario_name', 'Unknown')}
Annual Cost: {scenario_data.get('annual_cost', 'N/A')}

Generation per Asset:
{gen_lines}

Generation Costs per Asset:
{cost_lines}

Capacity Factors:
{cf_lines}

Based on these results, provide a brief (200 words max) critical analysis addressing:
1. Economic efficiency of the generation mix
2. System composition strengths/weaknesses
3. Key recommendations for improvement"""

        response = self.client.chat.completions.create(
            messages=[
                {"role": "system", "content": self.context_prompt},
                {"role": "user", "content": scenario_prompt}
            ],
            model="gpt-4o-mini",
            store=True,
        )

        return response.choices[0].message.content

    def create_markdown_report(self, scenario_data: Dict[str, Any], critique: str, results_root: str) -> None:
        """Create a markdown report for a single scenario"""

        now = datetime.now().strftime("%Y-%m-%d %H:%M")
        scenario_name = scenario_data.get('scenario_name', 'Unknown')
```

````python

        markdown = f"""# Scenario Analysis Report: {scenario_name}
Generated on: {now}

## Scenario Overview
![Scenario Comparison](scenario_comparison.png)

<div style="display: flex; justify-content: space-between;">
<div style="width: 48%;">

## Investment Analysis
- 10-year NPV: {scenario_data.get('npv_10y', 'N/A'):,.2f}
- 20-year NPV: {scenario_data.get('npv_20y', 'N/A'):,.2f}
- 30-year NPV: {scenario_data.get('npv_30y', 'N/A'):,.2f}
- Initial Investment: {scenario_data.get('initial_investment', 'N/A'):,.2f}
- Annual Operating Cost: {scenario_data.get('annual_cost', 'N/A'):,.2f}

</div>
<div style="width: 48%;">

## Generation Statistics

### Generation per Asset
```
{self._format_dict({k: v for k, v in scenario_data.items() if k.startswith('gen_')})}
```

### Generation Costs per Asset
```
{self._format_dict({k: v for k, v in scenario_data.items() if k.startswith('gen_cost_')})}
```

</div>
</div>

## Storage State of Charge
![Storage SOC Comparison](figure/storage_soc_comparison.png)

## Executive Summary
{critique}

---
"""
        # Create scenario folder if it doesn't exist
        scenario_folder = os.path.join(results_root, scenario_name)
        os.makedirs(scenario_folder, exist_ok=True)

        # Save markdown report
        report_path = os.path.join(scenario_folder, f"{scenario_name}_analysis.md")
        with open(report_path, 'w') as f:
            f.write(markdown)

        print(f"Analysis report saved to '{report_path}'")

    def _format_dict(self, d: Dict[str, Any]) -> str:
        """Helper function to format dictionary data for markdown"""
        return '\n'.join([f"{k.replace('gen_', '').replace('gen_cost_', '').replace('capacity_factor_', '')}: {v}"
                         for k, v in d.items()])

````

````python
    def _create_seasonal_comparison(self, scenario_name: str, results_root: str) -> None:
        """Create seasonal comparison plot"""
        scenario_folder = os.path.join(results_root, scenario_name)
        figure_folder = os.path.join(scenario_folder, "figure")
        os.makedirs(figure_folder, exist_ok=True)

        # Create figure with three subplots side by side
        fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(24, 6))

        # Plot each season
        for ax, season in zip([ax1, ax2, ax3], ['winter', 'summer', 'autumn_spring']):
            season_image = os.path.join(figure_folder, f'{season}_generation.png')
            if os.path.exists(season_image):
                img = plt.imread(season_image)
                ax.imshow(img)
                ax.axis('off')
                ax.set_title(f'{season.capitalize()} Generation')

        # Add overall title
        plt.suptitle(f'Seasonal Generation Comparison - {scenario_name}',
                     fontsize=16, y=1.02)

        # Save plot
        plt.savefig(os.path.join(figure_folder, 'seasonal_comparison.png'),
                    bbox_inches='tight', dpi=300)
        plt.close()

    def analyze_scenario(self, scenario_data, results_root):
        """Analyze a single scenario"""
        # Get scenario identifiers
        scenario_id = scenario_data['scenario_id']
        base_scenario = scenario_data['base_scenario']

        # Create output directory
        scenario_dir = os.path.join(results_root, base_scenario)
        os.makedirs(scenario_dir, exist_ok=True)

        # Rest of the analysis code...

    def _format_dict_as_table(self, d: Dict[str, Any], format_str: str = "{:,.2f}") -> str:
        """Helper function to format dictionary data as markdown table rows"""
        return '\n'.join([f"| {k} | {format_str.format(v)} |"
                          for k, v in d.items() if v and not pd.isna(v)])

    def create_global_comparison_report(self, all_scenarios_data: pd.DataFrame, results_root: str) -> None:
        """Create a markdown report comparing all scenarios"""
        now = datetime.now().strftime("%Y-%m-%d %H:%M")

        markdown = f"""# Global Scenarios Comparison Report
Generated on: {now}

## Investment Analysis

```
"""
        # Ensure numeric columns
        numeric_cols = ['npv_10y', 'npv_20y', 'npv_30y', 'annuity_30y',
````
````

````python
                     'initial_investment', 'annual_cost', 'annual_costs']
        for col in numeric_cols:
            if col in all_scenarios_data.columns:
                all_scenarios_data[col] = pd.to_numeric(all_scenarios_data[col], errors='coerce')

        # Sort by 30-year NPV
        sorted_scenarios = all_scenarios_data.sort_values('npv_30y', ascending=False)

        # Add header for full comparison table
        markdown += "Scenario".ljust(15)  # Reduced width for scenario number
        headers = ["Initial Inv.", "Annual Cost", "10y NPV", "20y NPV", "30y NPV", "Annuity"]
        for header in headers:
            markdown += header.ljust(20)
        markdown += "\n" + "-" * 125 + "\n"  # Adjusted length

        # Add data rows
        for idx, row in sorted_scenarios.iterrows():
            try:
                # Extract just the scenario number
                scenario_num = row['scenario_name'].split('_')[-1]
                markdown += (f"{scenario_num}".ljust(15) +  # Scenario number only
                           f"CHF {row.get('initial_investment', 0):,.0f}".replace(",", "'").ljust(20) +
                           f"CHF {row.get('annual_cost', 0):,.0f}".replace(",", "'").ljust(20) +
                           f"CHF {row.get('npv_10y', 0):,.0f}".replace(",", "'").ljust(20) +
                           f"CHF {row.get('npv_20y', 0):,.0f}".replace(",", "'").ljust(20) +
                           f"CHF {row.get('npv_30y', 0):,.0f}".replace(",", "'").ljust(20) +
                           f"CHF {row.get('annuity_30y', 0):,.0f}".replace(",", "'").ljust(20) + "\n")
            except (ValueError, TypeError):
                print(f"Warning: Invalid values for scenario {row['scenario_name']}")
                continue

        markdown += "```\n\n"

        # Add annual cost comparison plot with updated styling
        markdown += "## Annual Cost Comparison\n\n"

        plt.figure(figsize=(12, 6))
        # Set figure style
        plt.style.use('default')
        plt.rcParams['figure.facecolor'] = 'white'
        plt.rcParams['axes.facecolor'] = 'white'

        valid_data = all_scenarios_data.dropna(subset=['annual_cost'])
        # Extract scenario numbers correctly - look for scenario_XX pattern
        scenarios = []
        for name in valid_data['scenario_name']:
            # Extract XX from scenario_XX or scenario_XX_something
            match = re.search(r'scenario_(\d+)', name)
            if match:
                scenarios.append(match.group(1))
            else:
````

```
                scenarios.append(name)  # fallback
        costs = valid_data['annual_cost']

        # Create plot with styling
        ax = plt.gca()
        plt.bar(scenarios, costs)

        # Style the plot
        ax.spines['top'].set_visible(False)
        ax.spines['right'].set_visible(False)
        ax.spines['left'].set_color('black')
        ax.spines['bottom'].set_color('black')

        # Add grid
        plt.grid(True, axis='y', linestyle='--', alpha=0.7, color='grey')
        plt.grid(False, axis='x')

        plt.xticks(rotation=45, ha='right')
        plt.ylabel('Annual Cost (CHF)')
        plt.title('Annual Cost Comparison Across Scenarios')
        plt.tight_layout()

        cost_plot_path = os.path.join(results_root, 'annual_cost_comparison.png')
        plt.savefig(cost_plot_path, bbox_inches='tight', dpi=300)
        plt.close()

        markdown += f"![Annual Cost Comparison](annual_cost_comparison.png)\n\n"

        comparative_prompt = f"""Analyze the following scenarios data and provide a comparative analysis:
Scenarios Parameters:
{pd.read_csv('../data/working/scenarios_parameters.csv').to_string()}

Economic Comparison:
{sorted_scenarios[['scenario_name', 'initial_investment', 'annual_cost', 'npv_30y']].to_string()}

Key points to address:
1. Overall trends in cost effectiveness
2. Trade-offs between different generation mixes
3. Key success factors in the better performing scenarios
4. Recommendations for future scenario design

Limit the analysis to 400 words."""

        response = self.client.chat.completions.create(
            messages=[
                {"role": "system", "content": self.context_prompt},
                {"role": "user", "content": comparative_prompt}
            ],
            model="gpt-4o-mini",
            store=True,
        )

        markdown += response.choices[0].message.content

        # Save the report
        report_path = os.path.join(results_root, "global_comparison_report.md")
        with open(report_path, 'w') as f:
            f.write(markdown)
```

```

        print(f"\nGlobal comparison report saved to '{report_path}'")

    def _generate_report_content(self, scenario_data: dict) -> str:
        """Generate the markdown report content for a scenario"""
        scenario_name = scenario_data.get('base_scenario', scenario_data['scenario_name'])
        now = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

        # Format generation data - filter out nan values
        generation_data = {k: v for k, v in scenario_data.items()
                          if k.startswith('gen_') and not k.startswith('gen_cost_')
                          and pd.notna(v) and v != 0}
        cost_data = {k: v for k, v in scenario_data.items()
                    if k.startswith('gen_cost_') and pd.notna(v) and v != 0}

        # Generate critique using OpenAI
        critique = self.generate_critique(scenario_data)

        # Create markdown report
        markdown = f"""# Scenario Analysis Report: {scenario_name}
Generated on: {now}

## Overview
![Annual Summary](figure/annual_summary.png)

<div style="display: flex; justify-content: space-between;">
<div style="width: 48%;">

## Financial Analysis
| Metric | Value |
|--------|--------|
| Initial Investment | CHF {scenario_data.get('initial_investment', 0):,.0f} |
| Annual Operating Cost | CHF {scenario_data.get('annual_cost', 0):,.0f} |
| NPV (10 years) | CHF {scenario_data.get('npv_10y', 0):,.0f} |
| NPV (20 years) | CHF {scenario_data.get('npv_20y', 0):,.0f} |
| NPV (30 years) | CHF {scenario_data.get('npv_30y', 0):,.0f} |

</div>
<div style="width: 48%;">

## Generation Analysis

### Annual Generation by Asset Type
| Asset Type | Generation (MWh) |
|------------|-----------------|
{self._format_dict_as_table(generation_data)}

</div>
</div>

### Generation Costs
| Asset Type | Cost (CHF) |
|------------|------------|
{self._format_dict_as_table(cost_data, "{:,.0f}")}

## Storage State of Charge
![Storage SOC Comparison](figure/storage_soc_comparison.png)

## AI Critical Analysis
```

```
{critique}

---
"""
        return markdown

    def _format_dict_as_table(self, d: Dict[str, Any], format_str: str = "{:,.0f}") -> str:
        """Format dictionary as markdown table rows with Swiss number formatting"""
        rows = []
        for k, v in d.items():
            key = k.replace('gen_', '').replace('gen_cost_', '').replace('capacity_factor_', '')
            try:
                # Format number with Swiss style (apostrophes as thousand separators)
                value = format_str.format(float(v)).replace(',', "'")
            except (ValueError, TypeError):
                value = str(v)
            rows.append(f"| {key} | {value} |")
        return '\n'.join(rows)

# Color mapping for technologies
TECH_COLORS = {
    'Gas': '#1f77b4',      # Blue
    'Nuclear': '#ff7f0e',  # Orange
    'Solar': '#2ca02c',    # Green
    'Solar Storage': '#101',  # Purple (as requested)
    'Wind': '#9467bd',     # Purple
    'Wind Storage': '#102'  # Brown (as requested)
}

def plot_winter_summer_generation(data, ax):
    # ... existing setup code ...

    # Create bars with updated colors
    winter_bars = ax.barh(y_pos, winter_values,
                          color=[TECH_COLORS.get(tech, '#333333') for tech in techs],
                          alpha=0.8, label='Winter')

    summer_bars = ax.barh(y_pos, summer_values,
                          color=[TECH_COLORS.get(tech, '#333333') for tech in techs],
                          alpha=0.4, label='Summer')  # Reduced alpha for summer

    # Remove bar value annotations

    # Update legend
    ax.legend(loc='center right', bbox_to_anchor=(1.15, 0.5),
              title='Season', frameon=False)

    # Add clearer season labels
    ax.text(ax.get_xlim()[0], ax.get_ylim()[1], 'Winter',
            ha='left', va='bottom', fontsize=10)
    ax.text(ax.get_xlim()[1], ax.get_ylim()[1], 'Summer',
            ha='right', va='bottom', fontsize=10)

    # ... rest of plotting code ...
```

References

[1] Göran Andersson. Modelling and analysis of electric power systems. Lecture Notes 227-0526-00, ETH Zürich, Zürich, Switzerland, March 2004. Power Systems Laboratory, ETH Zürich.

[2] Mirko Birbaumer, Klaus Frick, Peter Büchel, and Simon van Hemert. Predictive modeling: Lecture notes, 2022. Unpublished lecture notes.

[3] John Forrest and Robin Lougee-Heimer. Cbc user guide. *COIN-OR Foundation*, 2018. Open-source mixed integer programming solver.

[4] IRENA. Renewable power generation costs in 2022. Technical report, International Renewable Energy Agency, Abu Dhabi, 2023.

[5] Andreas Klinkert, Peter Fusek, and Stephan Bütikofer. Linear and integer linear optimization, 2024. Lecture notes for MSE Module FTP Optimiz C.

[6] MATPOWER Development Team. Matpower documentation, 2024. MATPOWER: A MATLAB Power System Simulation Package.

[7] OpenAI. Openai api quickstart, 2024. API documentation and getting started guide.

[8] Stefan Pfenninger and Iain Staffell. Renewables.ninja, 2024. Web application for simulating renewable energy production.

[9] Allen J Wood, Bruce F Wollenberg, and Gerald B Sheblé. *Power Generation, Operation, and Control*. John Wiley & Sons, 2013.