Specialisation Project (VT2)
HS2024

Enhanced Platform for Investment Analysis

Mixed-Integer 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

July 20, 2025

Imprint

Project: Specialisation Project (VT2)
Title: Enhanced Platform for Investment Analysis
Author: Rui Vieira
Date: July 20, 2025
Keywords: mixed-integer linear programming, MILP, quantitative modeling, python, strategic planning, optimization, asset valuation, power-flow, platform, forecasting, energy trading

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.

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: mixed-integer linear programming, MILP, quantitative modeling, python, strategic planning, optimization, asset valuation, power-flow, platform, forecasting, energy trading

Contents

1 Introduction

1.1 Context & Motivation

Electricity powers everything: homes, factories, even the servers running the latest AI tools. The world's hunger for energy keeps rising, and the shift to renewables like solar and wind is only speeding up. But that switch brings headaches: solar and wind aren't predictable, and as more things go digital, unpredictable spikes can shake the whole grid. Recent blackouts have made us realize how dependent we are on the grid and often forget that internet is for now no existing without electricity. It's clear: we need to get better at planning for both the next hour and the next decade, while also pushing for the integration of renewables sources of energy into the grid.

1.2 VT1 Bottlenecks

In the first version (VT1), we built a model that handled power flow and investment, testing it on a handful of scenarios we set up by hand. The basics worked:

- Demand was fixed and didn't adapt.
- Each scenario had to be loaded and managed manually.
- The math (LP) let us solve problems, but skipped things like startup costs or on/off states for power plants.
- We had to glue things together with custom scripts.

This setup proved the idea, but it missed the mark when it came to real-world details— especially picking the best grid setup based on cost and performance.

1.3 Goals and Scope of VT2

VT2 takes a major step forward, tackling the main gaps from the first version with a more advanced and realistic approach:

- **Realistic operational modeling:** Instead of only using continuous variables, VT2 upgrades to mixed-integer linear programming (MILP). This lets us explicitly represent unit commitment—so power plants can switch on or off, model asset retirements and simulate maintenance. Once the logic implemented, we are a few code lines away of also introducing minimum up/down times, or start-up costs models but these were not implemented in this version.
- **Data-driven renewable integration:** the new version now incorporates historical and reanalysis weather data to generate realistic wind and solar production forecasts. This means we're not just simulating "average" days, but capturing the variability and uncertainty that challenge the grid. making it possible to stress-test decisions against extreme events, seasonal lows, and sudden surges in demand.
- **Modular, robust codebase:** The entire model has been refactored into a modular Python package using `Poetry`. Automated testing and continuous integration ensure that every change is checked for consistency and reliability. This not only speeds up development but makes it easier to extend—whether that's adding new assets, market mechanisms, or scenario types.

With these improvements, VT2 shifts from a simple prototype to a more robust planning and decision-support tool. It can now assess investment and operational strategies under high renewable shares, tight operational constraints, and uncertain future conditions—laying the groundwork for future features like demand response and market-clearing simulation.

2 Introduction, Context, and Literature Review

2.1 Context and Motivation

Electricity powers everything: homes, factories, and even the servers running the latest AI tools. Global energy demand is rising, and the shift to renewables like solar and wind is accelerating. But this transition brings new challenges—renewables are unpredictable, and increasing digitalization means more volatile and less controllable demand. Recent blackouts have exposed how dependent we are on the grid, and how easily disruptions cascade to other critical infrastructure, including the internet. This makes planning—both short- and long-term—and robust renewable integration more urgent than ever.

2.2 Modeling Approaches: From LP to MILP

The initial model (VT1) handled power flow and investment with a linear programming (LP) framework, using fixed demand, hand-crafted scenarios, and continuous variables for all decisions. This allowed proof-of-concept studies but missed essential real-world constraints: manual scenario management, the absence of unit commitment (on/off states, start-up costs), and limited operational realism.

The new version (VT2) adopts Mixed-Integer Linear Programming (MILP), where discrete (binary) variables enable us to model on/off states, asset retirements, maintenance cycles, and investment decisions in a unified, realistic manner [**?**, **?**]. MILP co-optimizes capital and operational expenditure (CAPEX and OPEX), preventing suboptimal investment timing or dispatch caused by treating planning and operation separately.

2.3 Data-Driven Forecasting and Python Ecosystem

Short-term forecasting of PV and wind is critical for grid operation. Methods range from simple persistence baselines to advanced ML models. We benchmarked several, ultimately using gradient boosting (XGBoost) for primary forecasting and SARIMA as a baseline [**?**, **?**]. This enables the model to capture realistic renewable variability and test robustness against extremes.

Python's mature ecosystem makes it the dominant platform for optimization and ML in energy systems. We leverage libraries such as docplex (for CPLEX integration), scikit-learn, XGBoost, and statsmodels. Commercial solvers (CPLEX, Gurobi) dramatically outperform open-source options for large MILPs, making them the default for this project [**?**, **?**].

2.4 VT2 Model Advancements

VT2 refactors the model into a modular Python package (using `Poetry`), with automated tests and continuous integration for reliability and extensibility. The new formulation supports high renewable shares, explicit unit commitment, and richer scenario modeling—setting the stage for further extensions such as demand response or market simulation.

3 Literature and Toolchain Review

3.1 Mixed-Integer Programming in Power-System Planning

Mixed-Integer Programming (MILP) is an improved version of Linear Programming (LP) that allows to solve problems with discrete variables for example whether a unit is on or off (unit commitment). Such binary decisions can not be modelled with pure linear programming. This allows to integrate investment (capital expenditure) and operational (dispatch, or operating cost) decisions into one optimization

framework. One can co-optimize the true least-cost solution that considers both capex and opex together. [**?**, **?**].

Formulating generation expansion planning (GEP) as an MILP captures the binary nature of building decisions (build vs. not build) and can include operational details like unit commitment. Despite being possible to include operation details such as minimum on/off time, rampe rates, etc. in the model, it was not done in this project. Not only is it more realistic but it also avoids suboptimal decisions that could arise from treating planning and operation separately.

3.2 Short-term PV/Wind Forecasting Methods

Short-term forecasting of photovoltaic (PV) or wind power is vital for grid operations. Approaches include:

- **Persistence/Empirical**: Simple baselines, e.g., assuming tomorrow equals today.
- **Physical/NWP**: Use weather forecasts and physical models for power prediction.
- **Statistical**: ARIMA/SARIMA and ARIMAX models learn from historical data and exogenous variables [**?**]. They are interpretable and data-efficient but limited for nonlinearities.
- **Machine Learning**: Methods like neural networks, SVR, and especially gradient boosting (e.g., XGBoost) capture complex patterns and often outperform statistical models when sufficient data is available [**?**, **?**, **?**]. Gradient boosting is noted for its accuracy and speed in PV/wind forecasting.
- **Hybrid/Ensemble**: Combine models (e.g., ARIMA+ANN) for improved robustness, though added complexity may not always yield better results.

Given these findings, we used gradient boosting (XGBoost) for forecasting, with SARIMA as a baseline.

3.3 Solver Landscape and Selection

Solving large MILPs requires robust solvers. Commercial options (CPLEX, Gurobi, Xpress) are state-of-the-art, offering fast solve times and reliability [**?**, **?**]. Open-source solvers (CBC, GLPK, SCIP, HiGHS) are free but generally slower and less robust. Benchmarks show Gurobi and CPLEX are typically 12–100x faster than CBC, and commercial solvers solve more instances to optimality [**?**].

For this project, CPLEX was chosen for the sake of understanding the solver and for having a academic license. Additionally, CPLEX offers a Python API (docplex), which made integration with our Python-based workflow straightforward

3.4 Python Ecosystem for Optimisation and ML

This interoperability and abundance of libraries is a major reason Python is so dominant in these fields today. Our literature review also confirmed that many recent research works in energy systems adopt Python for similar tasks, citing its balance of user-friendliness and powerful capabilities

Below is a list of the most relevant libraries for optimization and machine learning:

- For optimization, libraries like PuLP and Pyomo allow flexible model formulation and solver switching [**?**]. We used the CPLEX Python API (docplex) for direct integration.
- For ML, scikit-learn and XGBoost provide powerful tools for data processing and forecasting. Others librairies such as PyTorch, TensorFlow, and Keras are the reference ones for deep learning and neural networks [**?**].
- Statsmodels was used for time-series (SARIMA) modeling as a baseline model.

4 Problem Definition and Scope

This project extends the first-year LP framework in two directions:

- **MILP investment-dispatch model** – replaces the single-year LP with a multi-year mixed-integer formulation that chooses both what to build and how to run it, so CAPEX and OPEX are minimised in one pass.
- **Independent ML forecasting module** – delivers day-ahead predictions of the exogenous time series (load and variable renewables). It is stand-alone for now but provides the data that a rolling short-term optimisation would need.

This section outlines what is inside the study, what stays outside, and which indicators we will track.

4.1 Planning Horizon and System Boundaries

The scope of the study is defined by several key elements:

Strategic horizon: The model considers a user-defined list of years, normally between 1 and 10. This allows the optimisation to capture the timing of asset builds and retirements.

Operational resolution: Each year is represented by three typical weeks (winter, summer, spring-autumn). This approach keeps the MILP size modest while preserving essential seasonal detail.

Forecast horizon: The main forecasting target is 24 hours ahead, with additional checks at 48, 72, and 168 hours. This aligns with day-ahead operational needs and allows us to observe how forecast quality degrades with longer horizons.

Network footprint: The optimisation is run on a fixed test grid, including buses, lines, and assets. This lets us attribute changes in results to the model itself, not to changes in the underlying data.

Everything behind the connection point, as well as retail tariffs and ancillary services, is excluded from the study.

4.2 Decision Variables and Constraints

Variable	Type	Status
$b_{g,y},\ b_{s,y}$	binary	new – build decisions for generators g and storage s
$u_{g,y,t}$	binary	new – on / off for thermal generators
$p_{g,y,s,t}$	continuous	carried over – dispatch per generator
$p^{\text{ch}}_{s,y,s,t},\ p^{\text{dis}}_{s,y,s,t}$	continuous	carried over – storage charge, discharge
$e_{s,y,s,t}$	continuous	carried over – state of charge
$f_{l,y,s,t}$	continuous	carried over – DC line flow

New or changed constraint families include:

- **Build–lifetime link:** Sums of $b_{g,y}$ or $b_{s,y}$ set the installed flag for each year and forbid overlapping rebuilds.
- **Unit commitment:** Minimum up/down times, start costs, and ramp limits are tied to $u_{g,y,t}$.
- **CAPEX annuity:** Annual cost terms are based on build binaries and a capital recovery factor.
- **Storage relaxed final SoC:** The end-of-week state of charge may float within 10% of capacity to speed up the solve.

All other LP constraints—such as power balance, line limits, storage energy balance, and renewable profiles—remain in place, but now reference the new binary variables where needed. These elements turn the LP into a MILP, giving the solver the freedom to co-optimise when to invest and how to operate.

4.3 Forecast Horizon and Accuracy Targets

The forecasting module generates:

- **Primary target** – 24 hourly values for the next day for each time series.
- **Additional checks** – 48 h, 72 h and 7-day horizons to stress-test model degradation.
- **Features** – calendar flags, recent lags, rolling means, simple weather proxies.
- **Models** – linear baseline, random forest, gradient boosting, LSTM; all treated with identical split and scaling rules.

Accuracy targets (set after an initial back-test):

Metric	**Day-ahead target**
MAE	$< 5\%$ of mean load
RMSE	$< 7\%$ of mean load

Longer horizons have looser bands, logged but not optimised against.

4.4 Key Performance Indicators

The following indicators are tracked to assess the value of the MILP and the forecasting module:

- **Cost:** Total discounted system cost, including CAPEX annuities and weighted OPEX, is the main optimisation objective.
- **Solver:** MILP wall-clock time and optimality gap are reported for each run.
- **Forecast:** MAE and RMSE are tracked for each horizon and time series; lower values are better.
- **Robustness:** The number of hours with unmet demand or line overload is monitored and should be zero.
- **Transparency:** The share of total cost by asset class is reported to support sensitivity analysis.

5 Scope, Model Structure, and Key Performance Indicators

5.1 Scope and Boundaries

The study focuses on grid investment and operation under high renewable shares, explicitly modeling both long-term asset decisions and short-term operational constraints. The boundaries are as follows:

- **Strategic planning horizon:** Multi-year, typically 1–10 years, capturing asset builds, retirements, and their timing.
- **Operational resolution:** Each year is represented by three typical weeks (winter, summer, spring/autumn), balancing MILP tractability and seasonal fidelity.
- **Forecasting:** Day-ahead (24h) renewable and demand forecasting, with additional checks at 48, 72, and 168 hours to assess degradation.
- **System boundaries:** The model runs on a fixed test grid (buses, lines, assets), with retail tariffs and ancillary services excluded. Everything behind the grid connection point is out of scope.

5.2 Model Structure and Variables

The MILP jointly optimizes investment (CAPEX) and operational (OPEX) decisions:

- **Investment variables:** Binary decisions for whether to build generators ($b_{g,y}$) or storage ($b_{s,y}$) in each year.
- **Unit commitment:** Binary on/off states for thermal units ($u_{g,y,t}$), supporting future extensions (min up/down, start costs).
- **Operational dispatch:** Continuous variables for generator output, storage charging/discharging, state-of-charge, and DC line flow.

Key new/changed constraints:

- **Asset lifetime:** Enforces non-overlapping rebuilds and tracks active years.
- **Unit commitment:** Enforces commitment logic, linking minimum up/down and future start costs.
- **CAPEX annuity:** Investment cost spread using a capital recovery factor.
- **Storage:** Final state-of-charge relaxed within 10% of nominal, improving solver speed.

All core physical and economic constraints (power balance, line limits, storage, renewables) remain, now referencing new binary variables as needed.

5.3 Forecasting Module

A stand-alone ML forecasting module predicts exogenous series (load, wind, PV) with the following features:

- **Targets:** 24 hourly values for the next day, plus 48h, 72h, and 7-day checks.
- **Features:** Calendar indicators, recent lags, rolling means, weather proxies.
- **Models:** Linear, random forest, gradient boosting, LSTM (all use common splits/scaling).

Accuracy is quantified via MAE and RMSE, with day-ahead targets set to <5% (MAE) and <7% (RMSE) of mean load.

5.4 Key Performance Indicators (KPIs) and Assessment

The following metrics are tracked to evaluate the planning and forecasting framework:

- **System cost:** Total discounted cost (CAPEX annuity + OPEX) as main optimization objective.
- **Solver performance:** MILP wall-clock time and optimality gap for each scenario.
- **Forecast accuracy:** MAE and RMSE for each horizon and time series.
- **Robustness:** Number of hours with unmet demand or line overloads (should be zero).
- **Transparency:** Share of total system cost by asset class for scenario analysis.

These indicators validate the model's ability to recommend robust, cost-effective investment and operational strategies under uncertainty.

6 Linear to Mixed-Integer Programming Transition

This section builds directly on the DC Optimal Power Flow (DC-OPF) formulation and storage constraints presented in the previous semester project ("[vt1])"). While the core network and dispatch logic remain intact, we extend the model to include multi-year investment decisions using a Mixed-Integer Linear Programming (MILP) formulation. To fully appreciate this section, readers are encouraged to revisit the previous report, where the LP formulation is detailed.

The MILP formulation introduces binary variables to capture investment timing and operational availability of assets, enabling the model to answer: ¿ "When should we build or replace which generator or storage asset to meet future demand at minimal cost?"

The formulation maintains the original DC-OPF structure (Section **??**) and storage dynamics (Section **??**), but modifies and adds constraints to represent investment logic. The main changes are detailed below.

6.1 Methodology logic

i. Binary build and availability variables
Two 0/1 variables are introduced for every asset a (generator or storage) and planning year y:

$$\underbrace{b_{a,y}}_{\text{build}} = \begin{cases} 1 & \text{if the unit is } \textit{commissioned} \text{ in } y, \\ 0 & \text{otherwise,} \end{cases} \qquad \underbrace{i_{a,y}}_{\text{installed}} = \begin{cases} 1 & \text{if the unit is } \textit{operational} \text{ in } y, \\ 0 & \text{otherwise.} \end{cases}$$

These variables replace the fixed scenario-based asset list from the vt1 model.

ii. Lifetime-aware build–installed coupling
Each asset lives exactly L_a years. A sliding-window rule (see Eq. (**??**)) links the two binaries while preventing "double builds" inside its lifetime window (Eq. (**??**)).

iii. Capacity activation via installation flag
Generator dispatch p^{gen}, storage power ($p^{\text{ch}}, p^{\text{dis}}$) and energy soc are all *multiplied* by the installation flag $i_{a,y}$; if a unit is not built, its capacity is mathematically zero.

iv. Unified cost metric
Capital expenditure (CAPEX) is annualized using the Capital Recovery Factor (CRF). It is an asset-specific financial metric used to calculate the annualized cost of an investment over its lifetime. The goal is to make investment and operational costs comparable in a single linear objective. It is defined as:

$$\text{CRF}_a = \frac{i\,(1+i)^{L_a}}{(1+i)^{L_a} - 1}, \tag{1}$$

where i is the interest rate and L_a the lifetime of the asset.

v. Demand growth inside the model
To mimic a time-varying load through the years, the loads are scaled within the nodal-balance by a defined factor γ_y. This induces a growing demand in the model without increasing the generation. This should induce a diversification of the generation mix through the years.

6.2 Mathematical Implementation

For clarity the modifications are grouped into **(A) MILP-specific constraints** and **(B) code-level modelling improvements**. Classical DC-OPF balances and line limits remain unchanged. [**?**]

A. MILP integration new constraints

- **Build–Installed coupling** (NEW)
  Once built, a unit stays alive for L_a years; build at most once during that life.

$$i_{a,y} = \sum_{\substack{y' \le y \\ y - y' < L_a}} b_{a,y'} \qquad \sum_{\substack{y' \le y \\ y - y' < L_a}} b_{a,y'} \le 1 \tag{2a, 2b}$$

In plain terms:

- the first equation says "asset is operational in year y if built in one of the last L_a years."
- the second equation says "you can build or replace at most once per lifetime."

- **Generator Output Limits** (MODIF.)
  blablabla
$$0 \le P^{\text{gen}}_{g,y,s,t} \le P^{\text{nom}}_{g} \cdot \cdot i_{g,y} \tag{3}$$

- **Storage Power Limits** (MODIF.)
  blablabla
$$\begin{aligned} 0 \le P^{\text{ch}}_{s,y,s,t} &\le P^{\text{nom}}_{s} \cdot i_{s,y} \\ 0 \le P^{\text{dis}}_{s,y,s,t} &\le P^{\text{nom}}_{s} \cdot i_{s,y} \end{aligned} \tag{4}$$

- **Energy Capacity Limit** (MODIF.)
  blablabla
$$0 \le E_{s,y,s,t} \le E^{\text{nom}}_{s} \cdot i_{s,y} \tag{5}$$

  These replace the vt1 constraints where capacities were fixed and exogenously defined.

B. Additional modelling changes

- **Seasonal state-of-charge boundary** (MODIF.)
  The cycle-equality used in vt1 is relaxed.
$$\text{soc}_{s,y,s,0} = 0, \qquad 0 \;\le\; \text{soc}_{s,y,s,T} \;\le\; 0.1\, E^{\text{nom}}_{s}\, i_{s,y}. \tag{6}$$

  The battery starts empty and may end anywhere within 10 % of its energy capacity, avoiding cross-season infeasibility.

- **Nodal balance with load scaling** (MODIF.)
  For each bus b, season s, year y, hour t:
$$\sum_{g \in \mathcal{G}_b} P^{\text{gen}}_{g,y,s,t} + \sum_{s \in \mathcal{S}_b} (P^{\text{dis}}_{s,y,s,t} - P^{\text{ch}}_{s,y,s,t}) + \sum_{\ell \in \text{in}(b)} F_{\ell,y,s,t} = \gamma_y \cdot D_{b,s,t} + \sum_{\ell \in \text{out}(b)} F_{\ell,y,s,t} \tag{7}$$

  The only change is the growth factor γ_y on the demand term.

C. Objective function with annualised CapEx

The new cost function minimises both dispatch cost and annualised investment:

$$\min \quad \underbrace{\sum_{s \in \Sigma} W_s \sum_{y,t,g} c_g \cdot P^{\text{gen}}_{g,y,s,t}}_{\substack{\text{Operating Costs} \\ \text{OPEX}}} + \underbrace{\sum_{y} \Big(\sum_{a \in \mathcal{A}} A_a \cdot i_{a,y} \Big)}_{\substack{\text{Investment Costs} \\ \text{CAPEX}}} \tag{8}$$

With:

- W_s: Number of calendar weeks represented by each season s (e.g., winter = 13)
- $A_a = \text{CRF}_a \cdot C^{\text{cap}}_a$: Annualised CapEx per asset a

Hence the MILP simultaneously finds the least-cost *dispatch* and the cheapest *build / replace* schedule over the planning horizon. It eliminates the need for external spreadsheets computing NPV and manual scenarios comparison.

6.3 Implementation Details (MILP pipeline)

A. Overview and roles

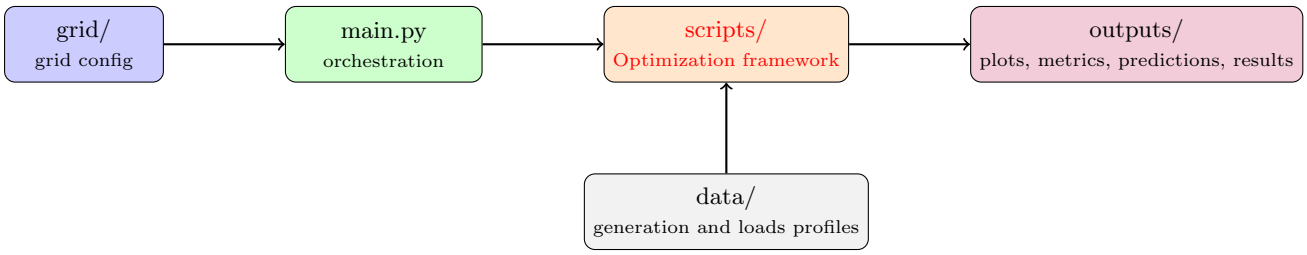


Figure 1: Compact overview of the forecasting pipeline.

- `main.py` – **driver script** accepting CLI flags, steering the four logical stages: preprocessing, network assembly, optimisation, and post-processing/logging.
- `pre.py` – slices three 168-h representative weeks, matches profiles to assets, and attaches analysis meta-data (years, season weights, load-growth).
- `optimization.py` – builds the **DC-OPF MILP** with annualised CAPEX, solves it via CPLEX, then serialises all variables back into the `IntegratedNetwork`.
- `post.py` – turns the raw decision variables into human-readable implementation plans, generation-mix graphics and asset timelines.
- `analysis/production_costs.py` – converts dispatch into MWh, adds annuitised investment streams, and prints/plots per-asset cost breakdowns.
- `results/` – central drop-zone for logs, JSON artefacts and figures.

B. Implementation

Figure 2 shows the deeper call graph inside the `scripts/` package, highlighting the *three* execution phases:

Phase 1 – Pre-processing `pre.py` converts raw CSV/time-series into `grid_data` + `seasons_profiles`. A light sanity-check now ensures the `representative_weeks` sum to 52; otherwise a default 13/13/26 split is injected.

Phase 2 – Object assembly `network.py` wraps every season in a `Network` (data-only) and records them inside a global `IntegratedNetwork`. Tweaks that proved essential:

- robust bus-ID matching (string vs. integer);
- automatic snapshot creation with length T;
- load-growth factors attached for later scaling.

Phase 3 – MILP formulation & solve `optimization.py` hosts two core functions: `dcopf()` builds the model, while `investement_multi()` (sic) solves, extracts and stores all variables. Key modelling choices:

- *at-most-one-build-per-lifetime* window ⇒ removes legacy multi-binary logic.
- Annualised CAPEX via CRF (`compute_crf`); operational costs weighted by season-weeks.
- No slack variables; nodal balances must close.
- Storage SoC forced to zero at each season edge, breaking cross-season energy loops.

Phase 4 – Post-processing `post.py` renders a Gantt-like timeline, seasonal generation mixes and an implementation plan, whereas `analysis/production_costs.py` computes MWh and $ flows, re-using the same CRF/discount helpers to stay consistent with the objective.

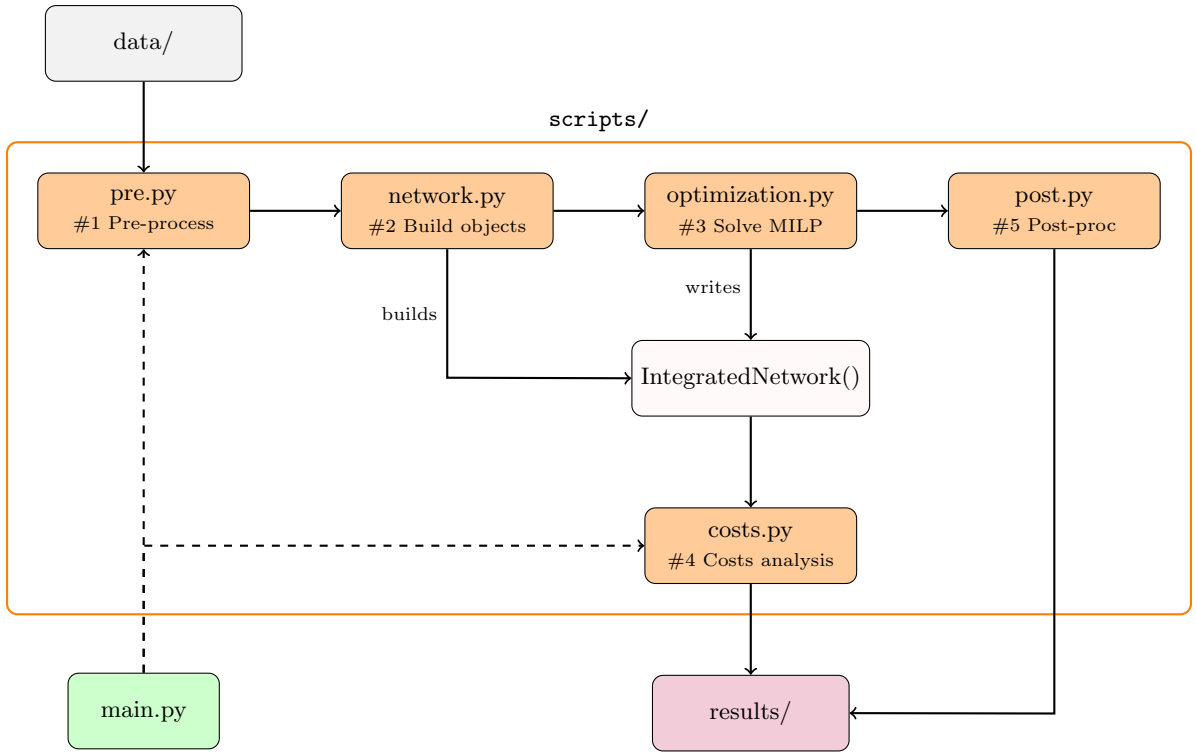
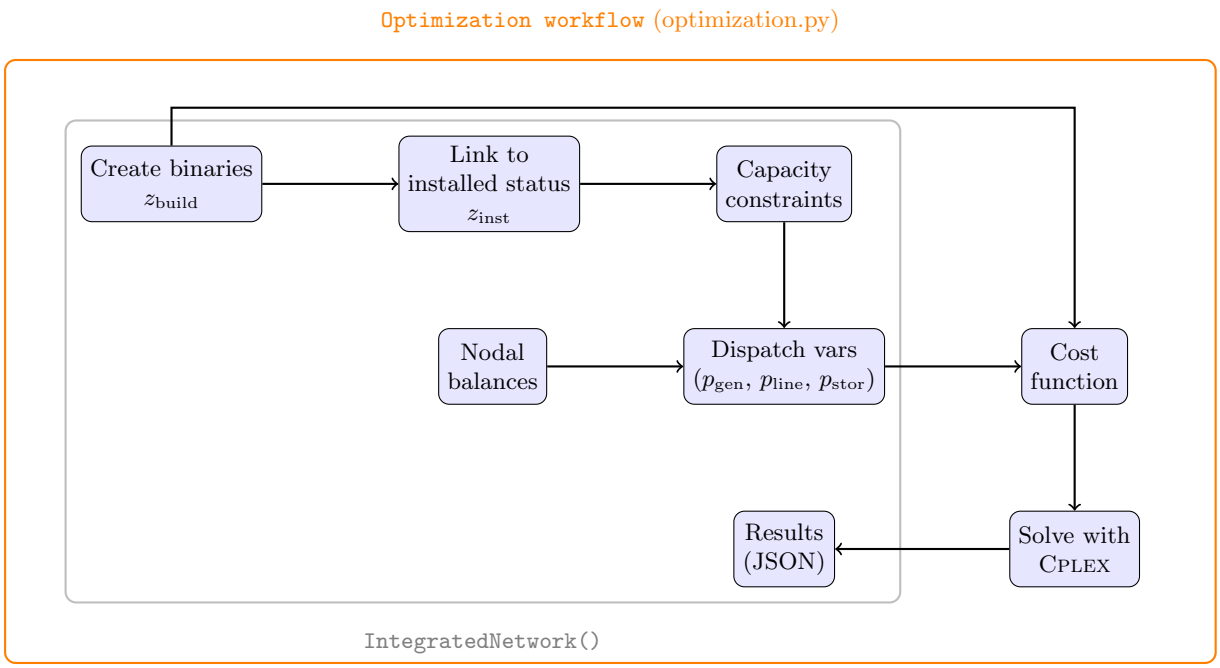


Figure 2: Detailed flow inside `scripts/`: preprocessing → object construction → MILP solution → reporting.

C. MILP model structure

Figure ?? sketches the internal control flow inside `dcopf()`. It shows how build binaries cascade into installed status, which then gate capacity and dispatch variables before everything is funnelled into a single cost function.



The resulting MILP stack therefore mirrors the earlier forecasting-pipeline layout while obeying power-

system specific lifetimes, energy conservation and investment logic.

7 Forecasting

This section walks through the full forecasting pipeline, starting with the *raw* data retrieved and ending with the **Gradient-Boosted Decision-Tree (GBDT)** model that ultimately ships to production. Each stage—data enrichment, feature engineering, model prototyping, and mathematical grounding—is documented so that a future engineer can reproduce (or challenge) every decision. The classical seasonal ARIMA (SARIMA) remains our statistical "baseline" and provides the first sanity check for all machine-learning attempts.

7.1 Data Enrichment

The investment model framework we previously developed offers **one year of hourly energy demand** that we opt to use as a static baseline and scale per asset. For the *forecasting* task, however, a richer data-context is essential.

1. **Meteorological history**
   We query the *Renewables.ninja* Point API (lat. 46.231°N, lon. 7.359°E, Sion) with the MERRA-2 re-analysis and retrieve **11 complete years** of hourly weather and PV simulation, January 2014 – December 2024.

2. **New data set**
   The original {`time`, `electricity`} table now includes temperature, rain rate, global and diffuse irradiance, cloud cover, and beam normal irradiance such as :

Table 1: API variables and physical meaning.

Symbol	Description	Unit
y_t	AC electricity output	kW
$t2m$	Air temperature @ 2m	
P_{rain}	Precip. rate (`prectotland`)	mm/h
$G_{\downarrow}$	Short-wave global irradiance (`swgdn`)	W/m^2
C_{tot}	Cloud-cover fraction (`cldtot`)	[0,1]
G_{dir}	Beam normal irradiance	W/m^2
G_{diff}	Diffuse irradiance	W/m^2
`T`	Air temperature	

   This means that we now have a data set with 9 features and 11 years of hourly data. The table now looks like this :

$$\{time, y_t,\ t2m,\ P_{\text{rain}},\ G_{\downarrow},\ C_{tot},\ G_{dir},\ G_{diff}, T, \}.$$

Some variables feed the model directly; others only serve as building blocks during feature engineering. Data quality was also assessments were also performed.

7.2 Feature Engineering

Feature engineering involves transforming raw time series data into informative inputs—such as lag values, rolling averages, or seasonal indicators—that help models capture patterns like trend and seasonality.

For SARIMA, features are built into the model via differencing and seasonal terms, while in machine learning, these engineered features explicitly guide the learning process to improve prediction accuracy. Features can be grouped into different groups :

- **Lagged target features**
  9 additional columns - Hourly self-lags capture short-term autocorrelation:

$$(y_{t-\ell},\ \ell \in \{1, 2, 3, 4, 5, 6, 12, 24, 168\}).$$

- fine-grain: $\ell \in \{1, 2, 3, 4, 5, 6, 12\}$ h
- diurnal: $\ell = 24$ h
- weekly: $\ell = 168$ h

Each lag adds one column $y_{t-\ell} = \texttt{electricity}(t - \ell)$. The longest history window is therefore one week.

- **Cyclical calendar encodings**
  6 additional columns - (sin, cos) for hour, weekday, year-day. Commonly called harmonics features:

$$\mathsf{hour\_sin}(t) = \sin\!\big(2\pi \tfrac{\mathrm{hour}(t)}{24}\big), \qquad \mathsf{hour\_cos}(t) = \cos\!\big(2\pi \tfrac{\mathrm{hour}(t)}{24}\big).$$

- **Weather interaction lags**
  14 additional columns - 7 vars $\times$ {1 h, 24 h}. To let the model learn delayed radiative effects we create one–hour and 24-hour lags for *every* meteorological column:

```
        weather_cols = ["t2m", "prectotland", "swgdn", "cldtot", "
            ↪ irradiance_direct",
        "irradiance_diffuse"]
    for col in weather_cols:
        df[f"{col}_lag_1"]  = df[col].shift(1)   # sensor latency
        df[f"{col}_lag_24"] = df[col].shift(24)  # diurnal memory
```

Listing 1: Python snippet – weather lag construction

- **Scaling**
  1 additional column - 7-day rolling mean of y_t used as trend anchor. Standardisation is applied *after* all lags are materialised to avoid data leakage:

```
    from sklearn.preprocessing import StandardScaler
    X_scaler, y_scaler = StandardScaler(), StandardScaler()
    X_scaled = X_scaler.fit_transform(X_raw) # predictor matrix
    y_scaled = y_scaler.fit_transform(y_raw[:,None])[:,0] # target
        ↪ vector
```

Listing 2: Python snippet – feature/target scaling

Resulting feature matrix. After dropping the first 168 rows (to satisfy the longest lag) and filling rare gaps by forward/backward-fill the design matrix contains

$$p = 37 \text{ predictors per hour, grouped as}$$

Backward Sequential Feature Selection (BSFS). Stage-2 BSFS removes entire *groups*, not individual columns, until the validation MAE stops decreasing. In practice *all five* buckets (lags, calendar, weather lags, trend scaler, raw weather) survive— confirmation that each family explains a distinct slice of variance.

7.3 Prototyping & Machine Learning Models

To evaluate the performance of the models we used the Mean Absolute Error (MAE) as a metric. A statistical benchmark SARIMA sets the baseline and every ML model should first beat it. Our original goal was a neural-network solution, so we iterated through increasingly sophisticated architectures but ended up switching to tree ensembles because of their interpretability insensitivity to feature scaling and overfitting.

Neural Network Models

1. **Simple Neural Network (NN)**
   A neural network is a computational model inspired by the human brain. In our model, it consisted of a single layer of interconnected nodes (neurons) that process data by applying weights, biases, and activation functions. Predictions are made from patterns learned during training.

   Bad: fails to capture strong seasonality → high bias.

2. **Multilayer Perceptron (MLP)**
   MLP is a specific type of neural network: a fully connected feedforward neural network with one or more hidden layers and nonlinear activation functions. We searched (tuned) for the optimal number of layers and units within an array of 1 to 4 layers × 1 to 128 units, trained on the same 37-dim matrix.

   Good: smooth forecasts; handles non-linearities.
   Bad: night-time over-fit despite target scaling tricks.

3. **Temporal Convolutional Network (TCN)**
   TCN is a type of neural network designed for sequential data. It uses 1D causal convolutions to capture temporal dependencies and are particularly effective for handling long-term dependencies.

   Good: tracks ramps well; long memory.
   Bad: 3× GPU time, sensitive to feature scaling, and still overshoots zero generation at night.

We explicitly set $y_t = 0$ between civil dusk and dawn (computed from the solar-zenith angle in the weather feed) to remove the night signal forecasting errors. Although this removed the worst negative bias, the best TCN still landed at MAE= 4.2% and required >90 min of hyper-parameter tuning. This was not a good trade off.

Pivot to Ensemble Trees

Gradient Boosted Decision Trees (GBDT) is an ensemble machine learning method. meaning in the end, all the trees are combined to make one powerful model where each new tree tries to reduce the errors (called Loss) made by the previous ones. This error is minimized via a technique called gradient descent who looks at how the error changes and then adjsuts the next tree based on it.

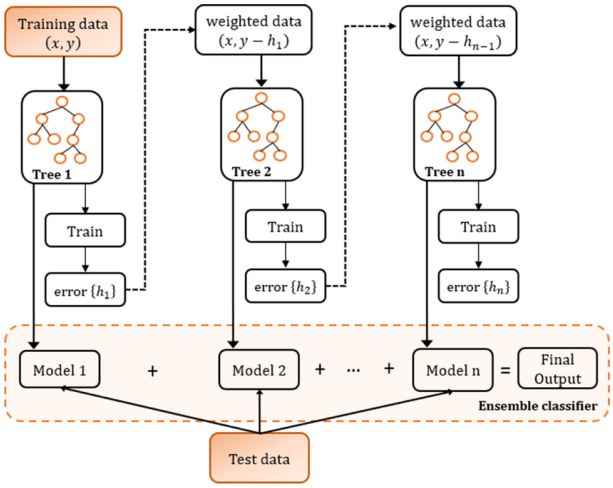


Figure 3: Gradient boosting decision tree (GBDT) illustration. Source: [**?**]

Contrary to Neural Networks, GBDT are then non-parametric, rule-based. They are then lighter and faster to train. The tunnable values are the number of trees, the depth of the trees, the learning rate and the regularization parameter.

The data is split into three parts: the training set, the validation set and the test set. The training set is used to train the model and the validation set is used to evaluate the model. The model is then evaluated

on the validation set and the process is repeated until the model is good enough. The model is then used to make predictions on the test set. The test set is used to evaluate the model.

GBDT are usually more interpretatabel and excel with tabular data (structured datasets). Howwever, GBDT does not inherently understand time or sequence, unless time features are manually engineered.

GBDTs use splits on feature values while TCNs use convolutions over time to model dependencies.

Despite the initial appeal of using neural networks like TCNs for their temporal awareness, the lack of familiarity with advanced libraries such as TensorFlow led to unsatisfying results. This prompted a return to the basics using scikit-learn and an exploration of GBDT, supported by literature highlighting its strengths.

GBDT emerged as the most suitable model due to its strong performance on tabular data, ease of capturing seasonality with calendar features, and superior accuracy and training efficiency compared to TCNs. Its minimal tuning requirements and further reinforced its suitability for the forecasting task.

7.4 Mathematical Underpinnings

Statistical Model - Baseline

We used SARIMA a statistical model as a benchmark. It applies ARMA modeling on a transformed (differenced) version of the time series to capture both short-term dynamics and repeating seasonal patterns. Its power lies in modeling both the temporal structure and seasonal cycles within a single, compact framework.

ARMA (p, q) is a linear combination of two things and works only on stationary data. It means the time series must have constant mean and variance over time. It is written as:

- an *autoregressive (AR)* part of order $p \rightarrow$ how past values influence the present
- a *moving average (MA)* part of order $q \rightarrow$ how past errors influence the present

The ARMA model is written as:

$$y_t = c + \sum_{i=1}^{p} \phi_i y_{t-i} + \varepsilon_t + \sum_{j=1}^{q} \theta_j \varepsilon_{t-j}$$

- c: constant term (mean of the process, if not differenced)
- ϕ_i: AR coefficients
- ε_t: white noise error term at time t
- θ_j: MA coefficients

SARIMA $(p, d, q) \times (P, D, Q)_s$ extends ARMA to handle trends and seasonality, in two ways:

1. **Differencing** for trend removal
   To remove non-stationary trends, SARIMA applies ordinary differencing d times:
   $$y'_t = (1 - L)^d y_t = y_t - y_{t-1} \quad (\text{if } d = 1)$$
2. **Seasonal differencing** for repeating patterns
   To remove seasonal effects (e.g., daily or yearly patterns), it applies seasonal differencing D times with period s:
   $$y''_t = (1 - L^s)^D y'_t = y'_t - y'_{t-s} \quad (\text{if } D = 1)$$

The six hyperparameters (p, d, q) and (P, D, Q) define the orders of autoregression, differencing, and moving-average smoothing at both the regular (hourly) and seasonal (daily) levels, with $s = 24$ reflecting the daily cycle. If the model residuals resemble white noise—i.e., they are uncorrelated and pattern-free—the model is considered to have successfully extracted all systematic, predictable structure.

Machine Learning Model - GBDT

Assume we aim to predict a target y from input features $x \in \mathbb{R}^n$. GBDT minimizes a loss function $L(y, \hat{y})$. (Minimizing mean absolute error via gradient descent between a predicted $\hat{y}$ and true y value).

Let:

$$F_0(x) = \text{initial guess} \qquad \text{for } m = 1 \text{ to } M$$

Boosting builds many shallow trees *sequentially*. Each tree tries to predict the *errors* the previous trees made:

$$r_i^{(m)} = y_i - F_{m-1}(x_i)$$

At each boosting step m, we compute the residual $r_i^{(m)}$, which is the difference between the true value y_i and the current model's prediction $F_{m-1}(x_i)$. This residual guides the new tree h_m to correct errors made so far.

The new model F_m is updated by adding a fraction $\nu \in (0, 1]$ (learning rate, controlling the step size) of the new tree $h_m(x)$ to the previous model's output. This sequential additive approach allows gradual improvement while avoiding overfitting.

$$F_m(x) = F_{m-1}(x) + \nu\, h_m(x)$$

After M trees, the final prediction is:

$$\hat{y} = F_M(x) = F_0(x) + \sum_{m=1}^{M} \nu \cdot h_m(x)$$

The final prediction $\hat{y}$ is the initial model $F_0(x)$ (often a constant like the mean of y) plus the sum of all M trees' predictions scaled by ν. This ensemble aggregates weak learners into a strong one.

Because every tree focuses on what is still unexplained, the ensemble gradually improves until additional trees no longer cut the validation loss.

Why GBDT wins here? With 37 engineered predictors the relationship to y_t is mostly *piece-wise* and *non-linear*. Rule-based splits excel at that, while requiring almost no architecture design—only {#trees, depth, ν} need tuning. The GBDT is then a good choice because it is non-parametric, rule-based, light and fast to train.

7.5 Implementation Details

This subsection provides a detailed description of the forecasting pipeline's code structure, its main components, and their interactions. The implementation is organized to streamline data ingestion, exploratory data analysis, feature engineering, model training with cross-validation, and evaluation.

A. Overview and roles

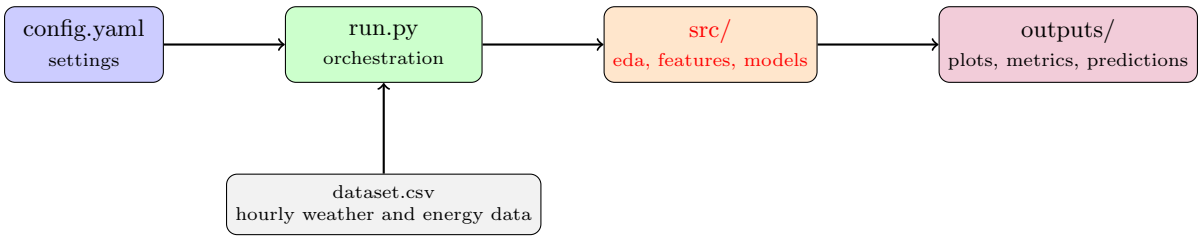


Figure 4: Compact overview of the forecasting pipeline.

- `config.yaml`
  acts as the central knobboard, holding configuration values such as dataset path, cut-off date for the test set, and cross-validation horizon.

- `run.py`
  controls the workflow in seven sequential steps, from loading data to model evaluation, while logging progress and writing output artefacts.

- `src/`
  contains the core logic with modules for exploratory data analysis (EDA), feature engineering, model training, and evaluation metrics.

- `outputs/`
  stores generated plots (e.g., heatmaps and pairplots) and evaluation reports (metrics and JSON summaries).

B. Implementation

The detailed flow inside `src/` is illustrated in Figure 5. The implementation distinguishes three main phases:

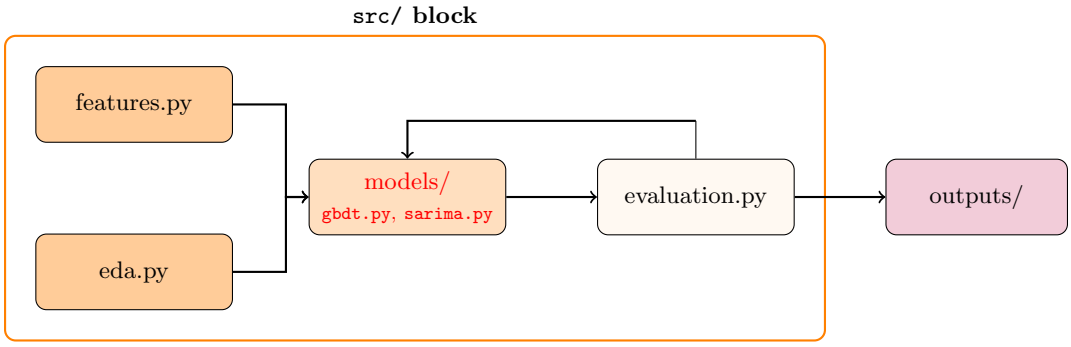


Figure 5: Final horizontal `src/` flow: feature engineering and EDA enter the `models/` block, with evaluation and feedback for optimization.

- `features.py`
  constructs engineered features including calendar-based sin/cos transforms, lagged variables, and rolling means to capture temporal dependencies.

- `eda.py`
  performs quick exploratory data analysis by saving visual summaries such as correlation heatmaps and pair plots for a sanity check on input data.

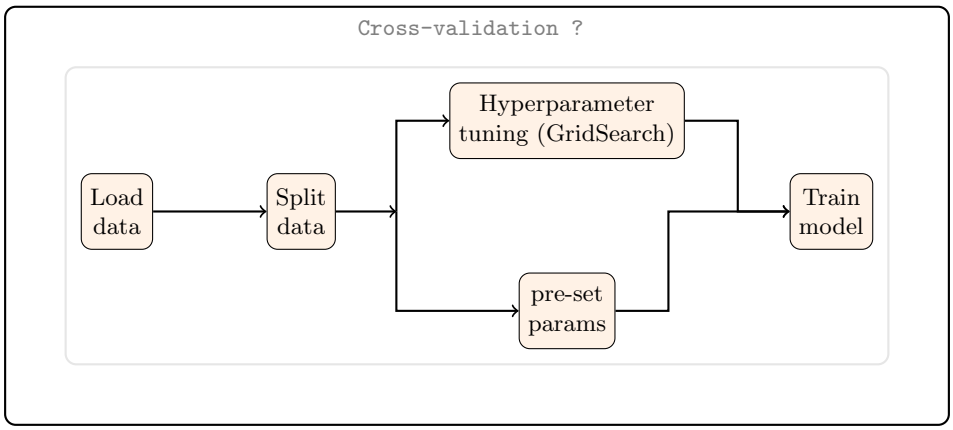
- The `models/` subfolder houses the modeling scripts (`gbdt.py` and `sarima.py`), where gradient boosting and SARIMA models are trained respectively.

- `evaluation.py`
  wraps common regression metrics like MAE, RMSE, and R^2, and formats output tables for CLI display.

The pipeline is designed with a feedback loop from evaluation back to modeling, allowing iterative tuning or refinement of models based on evaluation outcomes.

C. Models implementation

Figure 6 details the flow in `gbdt.py`, the main machine learning component. The process starts by loading and splitting the dataset into training and validation sets. Importantly, instead of a standard k-fold split, a *time series split* is used to preserve temporal ordering and avoid leakage across time in the validation procedure. This approach better reflects real-world forecasting challenges where future data is never accessible during training.



`gbdt.py`

Figure 6: Flow of `gbdt.py`: load and split data, explore default or tuned parameters, and train the model.

The model explores two training regimes: one with pre-set default hyperparameters, and another involving an explicit grid search over hyperparameters to find the optimal configuration. Both approaches culminate in training the final model on the training data.

Cross-validation is central to the feature selection and model tuning process. Several feature subsets are constructed and evaluated, including:

- **Manual**: A fixed set of manually chosen features.
- **Backward Sequential Feature Selection (BSFS)**: Starts from a large feature pool and iteratively removes less important features.
- **Forward Sequential Feature Selection (FwdSFS)**: Begins with a small seed set and adds features progressively.
- **Bayesian Optimization (BayesOpt)**: Searches over subset sizes k to optimize performance.

Each subset undergoes cross-validation using the `TimeSeriesSplit` strategy with multiple splits and a defined forecast horizon, ensuring robust evaluation respecting the temporal structure.

Figure **??** illustrates the SARIMA pipeline flow. Unlike the gradient boosting pipeline, SARIMA training does not require explicit splitting of the data, as the seasonal periods are incorporated during model specification. A grid search over seasonal orders (p, q, P, Q) identifies the best parameters based on the Akaike Information Criterion (AIC). After training, the model is evaluated on the test set, and results are exported for comparison.

Overall, the implementation balances between a fully automated pipeline and flexibility for manual adjustments and analysis, focusing on robust ML methods for time series forecasting.

8 Results

8.1 Forecasting Accuracy

Metrics overview

8.1.1 Forecast vs Actual Plot

Residual Diagnostics

Feature Model Info

CV Performance

Forecast Interval or Scenario Analysis

Table 2: Validation MAE by model (2022–2023 split).

Model	MAE [kW]	Rel. Δ vs. SARIMA
Naïve Seasonal	0.142	+34%
SARIMA baseline	0.106	—
MLP (2×128)	0.565	+433%
TCN (64f, 4 blk)	0.128	+21%
GBDT (XGBoost)	0.090	−15%
SARIMA + GBDT (ours)	0.085	−20%

8.2 MILP vs Legacy LP

8.3 Solver Impact (CPLEX vs GLPK)

8.4 Sensitivity and Scenario Analysis

8.5 Integrated Workflow Demo

9 Discussion

9.1 Interpretation of Key Findings

9.2 Trade-offs Analysis

9.3 Limitations

10 Conclusions and Outlook

10.1 Achievements Relative to Goals

10.2 Near-term Tasks

10.3 Long-term Vision

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.