Technology
Arts Sciences
TH Köln

M.Eng Automation and IT

Cyber Physical Energy Management

System Case Study

Jigisha Sanjay Ahirrao - 11222222
Kumpal Ganeshbhai Khokhariya - 11266110
Milankumar Hareshbhai Mavani - 11280466
Rishikesh Ravikiran Tiwari - 11257044
Niels Schwung - 11160885

May 5, 2025

Modelling and Simulation of Technical Systems (MSTS)

Contents

1 Project Description **1**
- 1.1 Project Scope . . . 1
- 1.2 Project Use Cases . . . 2

2 Obtaining Real Measurement Data **5**
- 2.1 Real World Measurement Devices Overview . . . 5
- 2.2 Siemens Sentron PAC 4200 . . . 5
- 2.3 Beckhoff Twincat . . . 6

3 Synthetic Data Generation in Simulink **8**
- 3.1 Simulated Industrial Profile . . . 8
- 3.2 MATLAB/Simulink Solver Configuration . . . 9
- 3.3 Controlled AC Power source with Harmonics . . . 11
- 3.4 Power Measurement and Calculation . . . 13
  - 3.4.1 Voltage Measurement . . . 13
  - 3.4.2 Current Measurement . . . 14
  - 3.4.3 Active and Reactive Power Measurement . . . 14
  - 3.4.4 Apparent power and Power Factor . . . 16
- 3.5 Configuration For Data Recording . . . 17
- 3.6 Load Selection . . . 18
- 3.7 Replicated Load Profiles . . . 19
- 3.8 Load Cycle Patterns . . . 20
- 3.9 Synthetic Data Generation for Load forecasting . . . 23

4 System Architecture: Simulation Execution and Data Handling **26**
- 4.1 Acquired Software and Data Flow . . . 26
- 4.2 HMI: Homepage . . . 28
- 4.3 Integration and Preprocessing in Python . . . 28
- 4.4 Matlab Function Code . . . 29
- 4.5 Database . . . 29
- 4.6 HMI: Simulation Page . . . 31

5 System Architecture: Load Forecasting **32**
- 5.1 Fundamentals of Load Forecasting: Role of Active and Reactive Power . . . 32
- 5.2 Model Selection: GRU . . . 33
- 5.3 Forecasting Algorithm: Model Training and Forecasting Flow . . . 33
- 5.4 HMI: Load Forecasting . . . 34

6 Implementation in the Laboratory Environment **37**
- 6.1 Software and Dependencies . . . 37
- 6.2 Folder Structure . . . 37
- 6.3 Steps to Run the System . . . 37
- 6.4 Notes on Laboratory Setup . . . 38

7 System Validation and Future Work **39**
7.1 Validation Criteria and Results 39
7.2 Cloud based Training of the Forecasting Network 39
7.3 Future Work 40

8 Conclusion **41**

A Appendix **I**
A.1 Full Python Code I
A.1.1 Python Code for Main Home Tab I
A.1.2 Python Code for Forecasting Tab III
A.1.3 Python Code for Simulation Tab XVII
A.2 Matlab Main Function Code XXVII
A.3 Multi-Output-Switch Code XXVIII

List of Figures

1 Modules of a CPEMS and their correlations. All objects within the Red frame are scope of this project. Source: [1] (picture edited) 1
2 Identified Use Cases of a CPEMS application with their baseline requirements (green), targeted results (yellow) and solution approaches (blue) 4
3 Live Power Measurement of an Electric Engine in the lab using the Siemens Sentron integral HMI 5
4 Example of Measurement Data acquired from the Siemens Power Config software 6
5 Program Code in Beckhoff Twincat used to obtain the measurement data format for current (Strom), voltage (Spannung) and power (Leistung) 7
6 Simulated Simulink Model 8
7 Solver Configuration in Simulink 10
8 3-phase Controlled AC power source in Simulink 11
9 Harmonics Signal for Voltage source 12
10 Three phase Voltage with and without Harmonics 14
11 Configuration of Power Measurement block 15
12 Power Measurement unit on each phase 17
13 To workspace block Configuration 18
14 Switch 18
15 Typical load curve of an industrial consumer over the day according to Tsekouras et. al. [2] 20
16 Polynom of degree 9 fitted to the modified industrial demand curve from Tsekouras et. al. [2] 21
17 Model of an exemplary weekly load cycle for a 1000 kW maximum consumption including random deviations based on the modified daily curve by [2] . . 22
18 Simulink step functions used for scheduling synthetic load profiles 24
19 End-to-end system flow diagram showing the integration between the Streamlit interface, MATLAB-based simulation engine, Python preprocessing, and Influxdb for time-series data storage and visualisation. 26
20 Data Flow between the Selected Program Infrastructure Elements of the Application (created with Napkin.ai) 27
21 CPEMS HMI Main Page – The user-friendly home screen provides direct access to forecasting and industrial simulation modules, supported by a clean layout and persistent sidebar navigation. 28
22 Code interaction between Python and Matlab) 30
23 Streamlit-based Simulation tab interface allowing users to select load parameters, execute MATLAB simulations in real time, and trigger automatic data preprocessing and storage into Influxdb. 31
24 Visualisation tab of the Streamlit interface displaying time-series data retrieved from Influxdb, including interactive plots of voltage, current, active/reactive/apparent power, and power factor for each phase. 31

25 Generate Forecasts Tab. Users can upload historical load data and generate energy forecasts for the next day or week using a pre-trained GRU model. Forecasts are visualised as time-series plots and can be downloaded for further use. 35
26 Train Model Tab. This interface allows users to retrain the GRU forecasting model on new data. Training progress is visualised with real-time loss plots, and the updated model can be saved for future forecasting tasks. 36

Abstract

This paper describes the implementation of a cyber-physical energy management system for industrial consumers. It covers the use cases of energy monitoring and load forecasting. It uses synthesized simulation data taken from a Simulink model that combines the industrial consumers of an induction motor, an induction furnace and a resistive load. The model also includes harmonics inserted via the power source and an industry typical daily load pattern. The simulation results are processed and written into an Influx DB database by the interaction of the Python main function, a Matlab function file and the Simulink model. A Gated Recurrent Unit (GRU) model implemented in a separate Python file is used to forecast future loads based on the historic simulation data on a daily and weekly basis. A Streamlit dashboard is implemented to allow the user to start the simulation, the training and the forecasting based on selected parameters and to visualize the results. The typical industry measurement outputs of Siemens Sentron and Beckhoff Twincat are taken as a reference for the data format to prepare the application for an implementation of real-world measurement data.

1 Project Description

1.1 Project Scope

Scope of the project is to develop a cyber physical energy managment system (CPEMS) with focus on industrial consumers. Below image 1 taken from [1] shows the elements covered framed in Red. The system is seen in overall context of all modules listed:

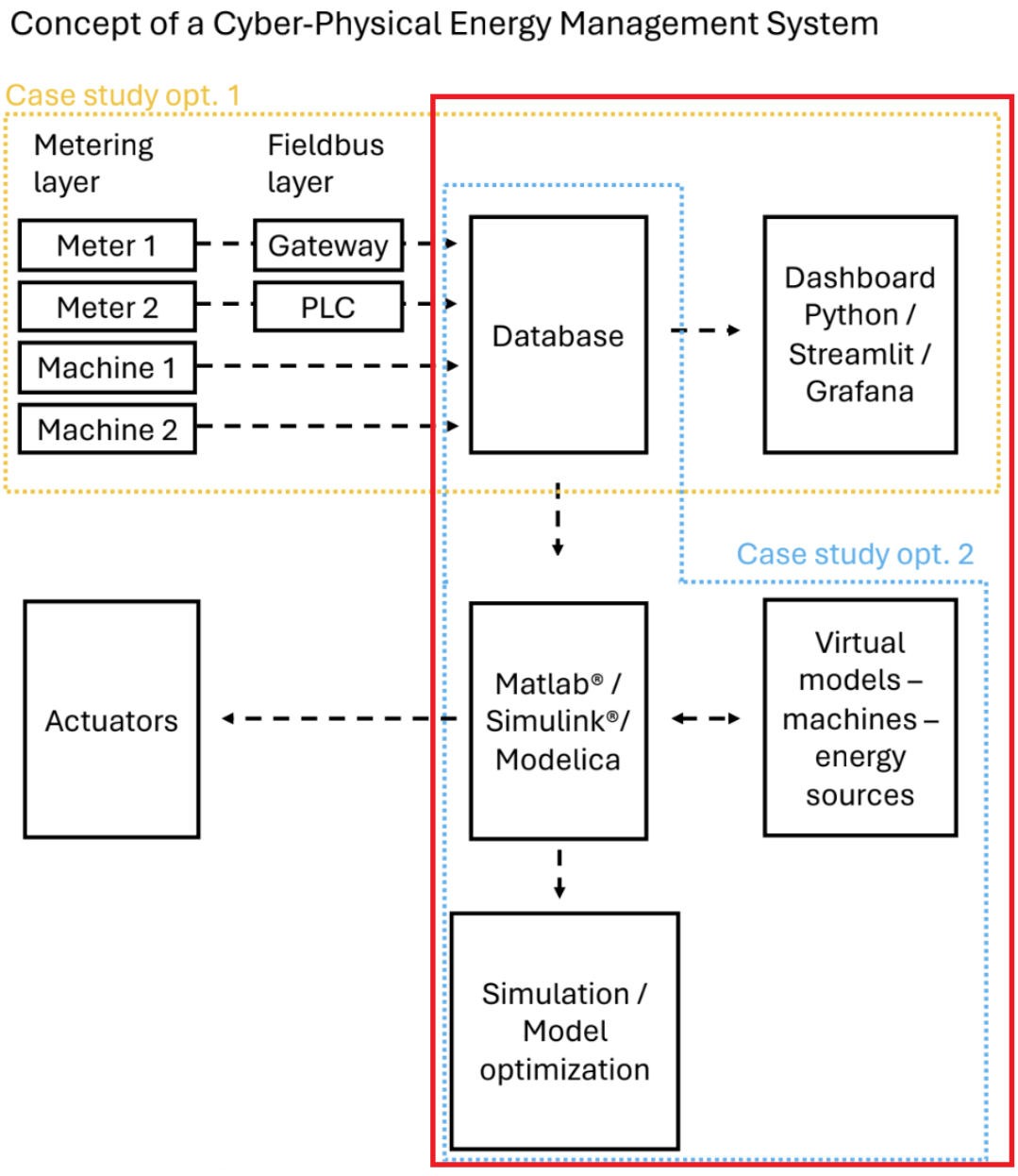


Figure 1: Modules of a CPEMS and their correlations. All objects within the Red frame are scope of this project. Source: [1] (picture edited)

Detailed measurement and detailed actuator analysis are not scope of this project. Key

elements include a database for data storage of real or synthesized data. Also a simulation of consumers such as actuators and energy sources including a forecasting algorithm are part of the project. In addition, a simple dashboard shall be programmed to make the data accessible and usable.

1.2 Project Use Cases

The most straightforward motivation for the implementation of a CPEMS at an industrial site is the straight knowledge about what is going on in terms of consumed amount of energy, but also in terms of a classification of the kind of machines that are currently in operation.

One mayor use case is the necessity of industrial companies to precisely measure their energy consumption to review such in energy audits. Besides own efforts to save energy, the motivation for such audits is increased in the European Union by legislation, such as the Energy Efficiency Directive (EED) 2012/27/EU, as part of the EU energy efficiency bundle, that in addition includes Ecodesign, Energy Labeling on consumer products and Energy performance in buildings. [3] According to the EED Article 8, Paragraph 2 [4], the

> "Member States shall develop programmes to encourage SMEs to undergo energy audits and the subsequent implementation of the recommendations from these audits. [...]"

,wherein the term "SME" stands for Small-Medium-Enterprise. Annex VI of the EED finally lists minimum criteria for such energy audits, which include for instance up-to-date data on energy consumption and electricity load profiles. Also, it mentions to build audits if possible on "life-cycle cost analysis instead of simple payback periods". Also, measures proposed during energy audits shall be underlaid with detailed and validated calculations on potential savings. [4] One other audit procedure is the ISO 50001. The target of such audits is the continuous improvement. The audits deal with the customer needs by looking at savings in the energy consumption from different view points like the technical view (production, enery use, ...) and the economic view (monetary amortization, etc.) [5].

However, not only political regulations, but also economic interests may increase the necessity of measuring the electricity consumption: For instance, public investments may be bound to energy efficiency performance, such as suggested in EED Article 6 [4]. Also, the Environmental Social Governance bound investments show an increasing trend on global finance markets: According to the National Bureau of Economic Research, in the U.S. the share of index funds with ESG mandate rose from 3 percent in 2019 to 5 percent in 2022 [6]. According to the research institute Statista, the Global ESG ETF assets reached 480 Billion US Dollars in 2023, which is an All Time High [7]. Therefore, acquired energy consumption data could support the certification for ESG criteria and therefore could help to acquire new investors or credits.

Still, the companies' ability of a digital tracking of electricity consumption is not fully given across the economy: For instance, [8] mentions a survey conducted by the German government-owned credit institute KfW among 9,500 German companies. Quoting the medium size company expert Elisabeth Grewenig, currently only 30 percent of the medium size businesses could [precisely] provide their electricity demand.

Although electricity is not always the primary form of energy, the application described herein is not covering direct emissions in terms of green house gases from chemical processes as dealt with in the EU Emissions Trading System (ETS) 2003/87/EC [9]. The relevant industries listed in Annex I of this directive, are explicitly excluded by the EED in article 7, paragraph 2 point b). [4]

The target variable of the electricity monitoring shall be complex power. In order to calculate the power from the measurement variables voltage (U) and current (I) on each of the three phases, mathematical relations are needed. By knowing the active power and the ration of active and reactive power, the energy demand can be taken as the most impactful cost driver. However, also the power ratio going out of the range agreed with the energy provider can cause additional cost.

By considering the best and hands-on knowledge on the current status, also a classification of the consumers that are in operation based on the measured load profiles is another possible use case. Such could be done in order to detect the most demanding machines and further to switch them off if they are idling, for instance. To perform the classification, the typical load patterns of available machines need to be compared with the current demand pattern. Such could be for instance done with some kind of time series classification algorithm.

In order to best make decisions not only for the current situation but also as a predictive choice, a load forecasting is key. Such can be obtained by using a time series forecasting algorithm, such as a neural network or similar. It should be capable to consider daily and weekly cycles as they typically appear in industrial shift models.[2] The historical loads will therefore be an input to forecast the future load. Clearly, the time scale of the forecast does not need to be as granular as the current measurment; a downscaling of the historical data is recommended.

Such an targeted switching action for current of future demands would match another use case and target of energy audits: Optimizing the energy efficiency and load consumption of a plant. Potentially, an advanced algorithm could serve as an expert system to advice the customer with typical optimization features applicable to its current situation. It could compare the current consumption and the forecast with desired performance indicators, such as a good power balance, other power quality metrics and efficient use of the consumed energy.

Following up the sustainability idea, adding consumers or renewable energy sources is another a possible use case. Whereas the adding of new or changed consumers is common in the evolution of business, the installation of own power generation such as photovoltaic is a consideration that gains further momentum in recent times. Increasing energy prices can be a motivation as well as environmental constraints or green label marketing purposes. In addition to the above, also governmental guidelines are being set in place making the energetic usage of roof top capacities mandatory: One example out of many is the GV.NRW.S.332 which requires a certain amount of the roof surface to be equipped with photovoltaic for new and existing renovated building roofs.[10] A CPESM application therefore could show the potentials of a photovoltaic generation with or without a battery storage in terms of its effect on the current electricity consumption. Related to the PV and battery integration, another

consideration might be the analysis of power outage vulnerability. In case a renewable energy generation and or a battery buffer are installed, the key question here is how long can key production operations be kept running without external energy supply. Load shedding can have significant impacts on the production of almost any good, even agricultural products, such as seen in 2023 in South Africa's wine industry. [11]

Whereas efficiency considerations focus on the integration of renewable energy sources, from economic point of view also a capacity analysis for any operation extension is of interest. Such could be for example the purchase of a new, high load consuming industrial machine or a device that bears the potential to impact the balance of active and reactive power.

Image 2 lists the mentioned use cases. It also shows the baseline requirements (second row, green) and the results (third row, yellow) that are expected to be obtained by a pre selection of possible algorithms and solution approaches (fourth row, blue). Clearly, the use cases build up on each other in the way that some take the results of another on as baseline. Therefore, it turns out that load displaying and load forecasting are the most important ones forming the foundation for the other, more advanced ones. Therefore, this project will cover these two only (framed red in figure 2. The other use cases may be subject to future work.

Energy Consumption	Machine Classification	Load Forecasting	Load Optimization / Efficiency	System Expansion	PV Generation
Synthesized Data of U and I	Industrial Load Patterns, Available Machines	Historical Data Pattern	Consumption and Capacity of each Machine, Typical Improvements	Expansion Plans, Power Supply Limits	Required Capacity, poss. Generation Patterns
Power Factor, Reactive/Active, Losses, THD	Confirmation of consumption pattern, Machines in OP	Future Load Pattern	Desired PF, P-Q-Balance, Best Improvement	Expanded Load Pattern, Break Limits y/n	Match Demand and Generation, Required Installation
Mathematics	Time Series Classification to each Machine	NN-Time Series or simpler algorigthm	Global Optimizer on Power and P-Q-Balance	Sum all Demands	Fitting Gen. Options to find the Best

Figure 2: Identified Use Cases of a CPEMS application with their baseline requirements (green), targeted results (yellow) and solution approaches (blue)

2 Obtaining Real Measurement Data

2.1 Real World Measurement Devices Overview

In order to build the generation of synthesized data close to reality, two industrial systems for power measurement are analyzed: The Siemens Sentron measurement with the Siemens Power Config software as well as the Beckhoff Twincat software. In the lab of the university, an electric engine is connected in a three phase setup to the measurement unit, running on neutral load.

2.2 Siemens Sentron PAC 4200

The Siemens Sentron device comes with an own HMI, as shown in picture 20. The measured data can be captured in a table format using the Siemens Power Config software; such an obtained measurement result is shown in figure 22. For every of the three lines, the current and the phase angle are plotted. However, the sample rate used in this example is not capable to detect the true waveform of the curve, which is 50 Hertz, as visible with the last timestamp; the sample rate would need to be more than 100 Hertz according to the common Shannon-Nyquist criterion. For power systems operating at 50 Hz (as in Europe), a sample rate of at least 100 Hz is the bare minimum to meet the Shannon-Nyquist criterion. However, in industrial applications, much higher rates (e.g., 1 kHz or more) are often preferred to capture harmonics, transients, and other power quality issues. Devices like Siemens Sentron often support industrial communication protocols (e.g., Modbus, PROFINET) allowing integration into SCADA or ERP systems for real-time monitoring and historical data logging. Measurement data from tools like Siemens Power Config can often be exported in standard formats (CSV, XML, etc.), enabling integration with databases or further processing in software such as MATLAB or Python for analysis.[12]

Figure 3: Live Power Measurement of an Electric Engine in the lab using the Siemens Sentron integral HMI

2	Timestamp	Current L1	Current L2	Current L3	Cosϕ L1	Cosϕ L2	Cosϕ L3
3	12/12/2024 13:27	774.2 mA	769.8 mA	788.1 mA	0.22	0.24	0.24
4	12/12/2024 13:27	774.0 mA	768.5 mA	786.8 mA	0.22	0.24	0.24
5	12/12/2024 13:27	774.7 mA	768.1 mA	785.6 mA	0.22	0.24	0.24
6	12/12/2024 13:27	773.4 mA	768.8 mA	787.3 mA	0.22	0.24	0.24
7	12/12/2024 13:27	773.9 mA	769.3 mA	787.5 mA	0.22	0.24	0.24
8	12/12/2024 13:27	773.4 mA	770.5 mA	788.8 mA	0.22	0.24	0.24
9	12/12/2024 13:27	772.7 mA	769.2 mA	788.6 mA	0.22	0.24	0.24
10	12/12/2024 13:27	773.5 mA	768.9 mA	788.2 mA	0.22	0.24	0.24
11	12/12/2024 13:27	774.3 mA	770.4 mA	789.0 mA	0.22	0.24	0.24
12	12/12/2024 13:27	773.5 mA	768.5 mA	788.0 mA	0.22	0.24	0.24
13	12/12/2024 13:27	773.4 mA	768.8 mA	788.3 mA	0.22	0.24	0.24
14	12/12/2024 13:27	774.6 mA	770.4 mA	790.5 mA	0.22	0.24	0.24
15	12/12/2024 13:27	773.1 mA	770.2 mA	787.8 mA	0.22	0.24	0.24
16	12/12/2024 13:27	772.8 mA	769.7 mA	787.8 mA	0.22	0.24	0.24
17	12/12/2024 13:27	774.5 mA	771.0 mA	788.7 mA	0.22	0.24	0.24
18	12/12/2024 13:27	772.1 mA	770.0 mA	786.8 mA	0.22	0.24	0.24
19	12/12/2024 13:27	772.8 mA	769.3 mA	787.0 mA	0.22	0.24	0.24
20	12/12/2024 13:27	773.5 mA	769.8 mA	787.4 mA	0.22	0.24	0.24
21	12/12/2024 13:27	772.2 mA	769.4 mA	787.0 mA	0.22	0.24	0.24
22	12/12/2024 13:27	773.0 mA	770.1 mA	788.2 mA	0.22	0.24	0.24
23	12/12/2024 13:27	773.4 mA	769.9 mA	787.1 mA	0.22	0.24	0.24
24	12/12/2024 13:27	772.9 mA	769.1 mA	787.8 mA	0.22	0.24	0.24
25	12/12/2024 13:27	774.2 mA	770.4 mA	788.2 mA	0.22	0.24	0.24
26	12/12/2024 13:27	773.2 mA	769.6 mA	787.2 mA	0.22	0.24	0.24
27	12/12/2024 13:27	774.1 mA	768.7 mA	787.3 mA	0.22	0.24	0.24
28	12/12/2024 13:27	776.7 mA	770.7 mA	789.9 mA	0.22	0.24	0.24
29	12/12/2024 13:28	776.4 mA	771.6 mA	790.1 mA	0.22	0.24	0.24

Figure 4: Example of Measurement Data acquired from the Siemens Power Config software

2.3 Beckhoff Twincat

The Beckhoff Twincat software is real time capable and therefore offer advanced capabilitie to detect true waveform curve. The sample rate can be defined by the user, so clearly for this application more than 100 Hertz are required. Unfortunately, some caveats were discovered when working with the Beckhoff program, especially with the memory limitations of the computation hardware available, which was one of the reasons to switch to the use of synthesized data instead. Also not every machine type handled was available in the lab for power measurement. However, the simulation of the true waveform data will try to come close to the Beckhoff data format. Image 5 shows the program used to obtain the format of the measurement data. Clearly, the sample rate can be defined by the user.

We attempted to generate and log real-time data using the Beckhoff Industrial PC C6030 along with its I/O modules. Our goal was to extract the operational data into a structured

CSV file for further analysis. While the device setup and programming were completed, we ultimately faced challenges in accessing the final CSV data output, and were unable to retrieve the complete dataset as intended.[13]

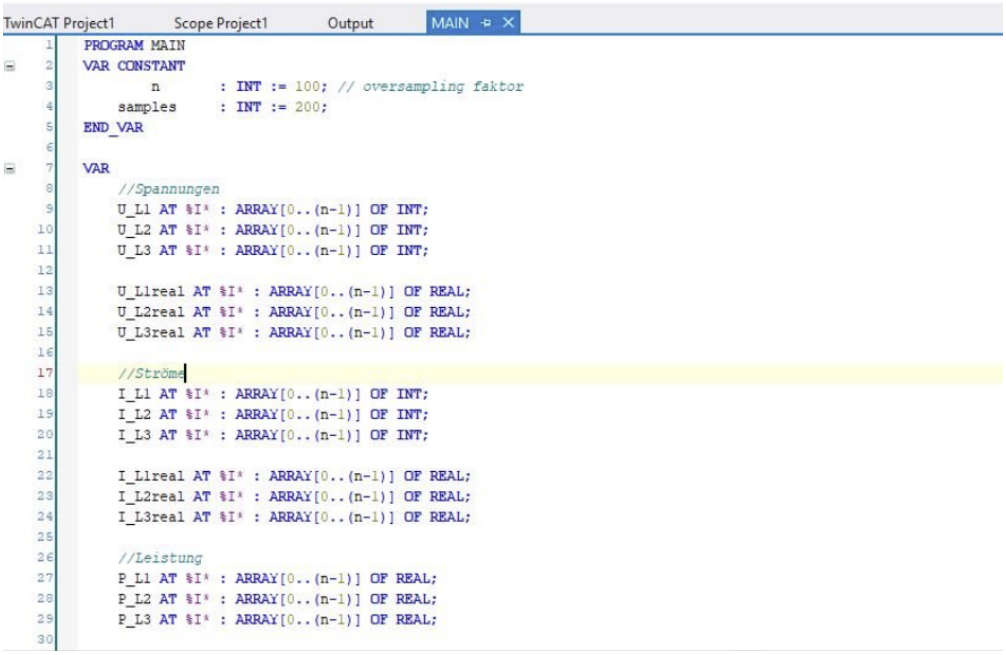


Figure 5: Program Code in Beckhoff Twincat used to obtain the measurement data format for current (Strom), voltage (Spannung) and power (Leistung)

In summary, there are two forms of data: True wave form format that includes the rough measurements of voltage, current, and phase angle, and an aggregated format that includes calculated values such as active power, reactive power, and power factor. Consequently, the overall model should include both of these.

3 Synthetic Data Generation in Simulink

3.1 Simulated Industrial Profile

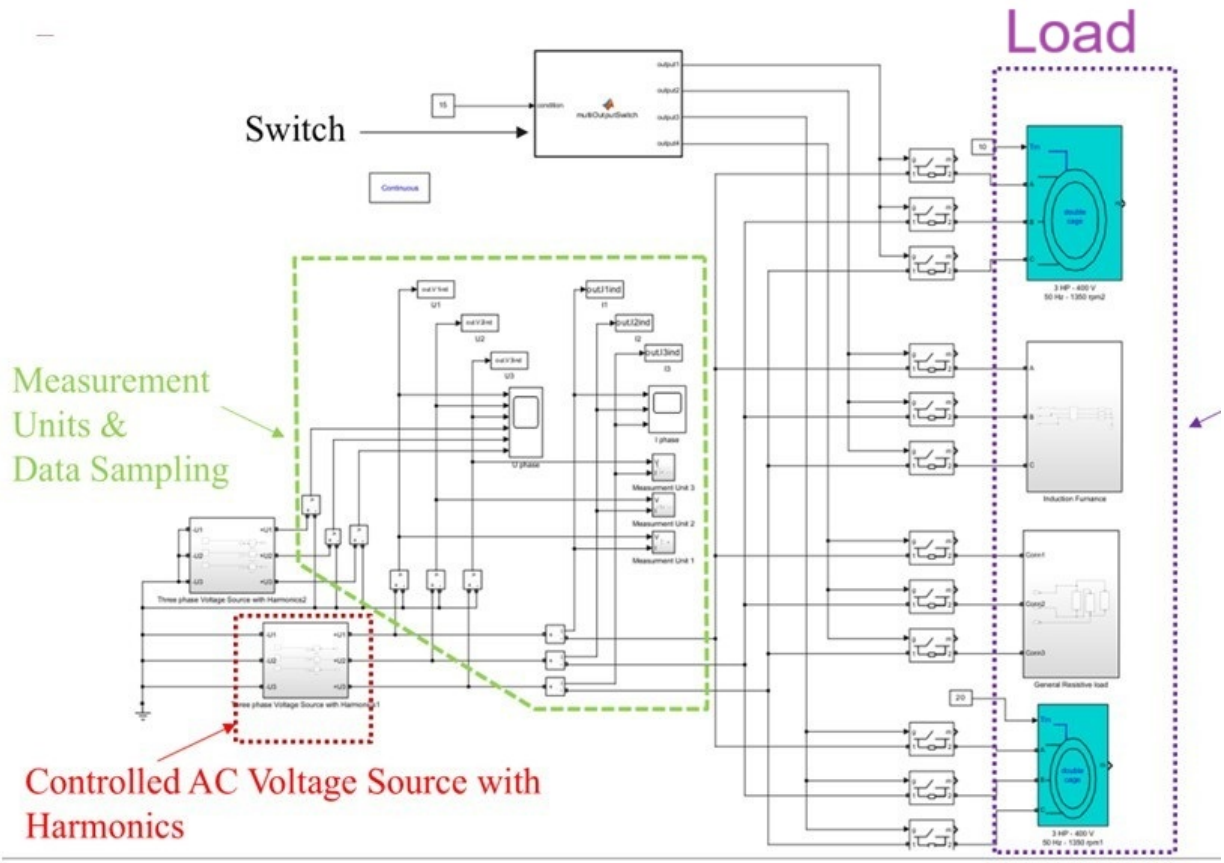


Figure 6: Simulated Simulink Model

The simulation model illustrated in Figure 6 was developed using MATLAB/Simulink to analyze the behavior of various load profiles commonly found in industrial environments. During the simulation process, several practical scenarios were considered, including the presence of harmonic distortion and the accurate calculation of key electrical parameters such as power, voltage, and current. The model is designed to evaluate load performance when subjected to a controlled three-phase AC voltage source that introduces harmonic components.

An important consideration in such analyses is the distinction between linear and nonlinear loads[14]. In general, waveforms provide a visual comparison: for linear loads, the current waveform closely resembles the voltage waveform—both exhibit clean 50 Hz sine waves. In contrast, nonlinear loads produce current waveforms that differ significantly from the voltage waveform due to harmonic content. These harmonics originate from the load itself and result in current distortion, which in turn leads to voltage distortion across the system. Devices such as variable frequency drives or electronic power supplies often contribute to

such nonlinearity.

However, in the present simulation model, nonlinearity effects were not observed. The modeled loads exhibit linear behavior, where the current waveforms maintain a consistent sinusoidal shape similar to the voltage waveform, indicating minimal harmonic-induced distortion. This allows for clearer analysis of harmonic injection from the voltage source itself rather than from the load characteristics.

1. **Controlled AC Voltage Source with Harmonics**
   Positioned at the bottom-left of the model (highlighted in red), this block serves as the input power supply. It is configured to inject harmonics into the three-phase voltage signal, allowing analysis of how loads behave under distorted voltage conditions. This setup mimics real-world power systems where harmonics are often present.

2. **Measurement Units and Data Sampling**
   Located within the green highlighted area, this section includes voltage and current measurement blocks for all three phases. The signals are sent to data sampling units, enabling real-time monitoring of key electrical parameters. These include voltage, current, active power, reactive power, apparent power, power factor, and waveform characteristics.

3. **Switching Block**
   Found at the top-middle of the model, this block acts as a control logic switch that allows the user to selectively activate different load configurations. It ensures that specific loads or combinations thereof operate under defined simulation conditions, enabling systematic profiling of load responses.

4. **Various Loads**
   Shown on the right side of the model (highlighted in purple), this section consists of several types of electrical loads connected to the system. These include:

   - A motor load (top)
   - An induction furnace
   - A general resistive load
   - A pump (bottom)

   Each load is equipped with general measurement unit to capture voltage, current, and power data during simulation runs.

3.2 MATLAB/Simulink Solver Configuration

Solver settings are vital, especially when simulating electrical load profiles, because electrical systems often exhibit rapid, discontinuous behavior such as switching operations, load connection or disconnection, and fast transient responses. Using a variable step solver helps in capturing fast transients without missing critical events, maintaining numerical stability, especially during switching operations, and enabling efficient simulation by automatically

adjusting the step size based on system demands, thereby reducing unnecessary computation during steady-state periods. Therefore, the chosen solver configuration provides a good compromise between simulation accuracy, stability, and computational efficiency.

In this simulation, a variable step solver is used to accurately capture the dynamic behavior of various electrical load profiles. The solver type is set to automatic selection, allowing Simulink to choose the most appropriate solver based on the system's characteristics. This flexibility ensures numerical stability and effectively handles the non-linearities and switching events that are commonly present in electrical systems.

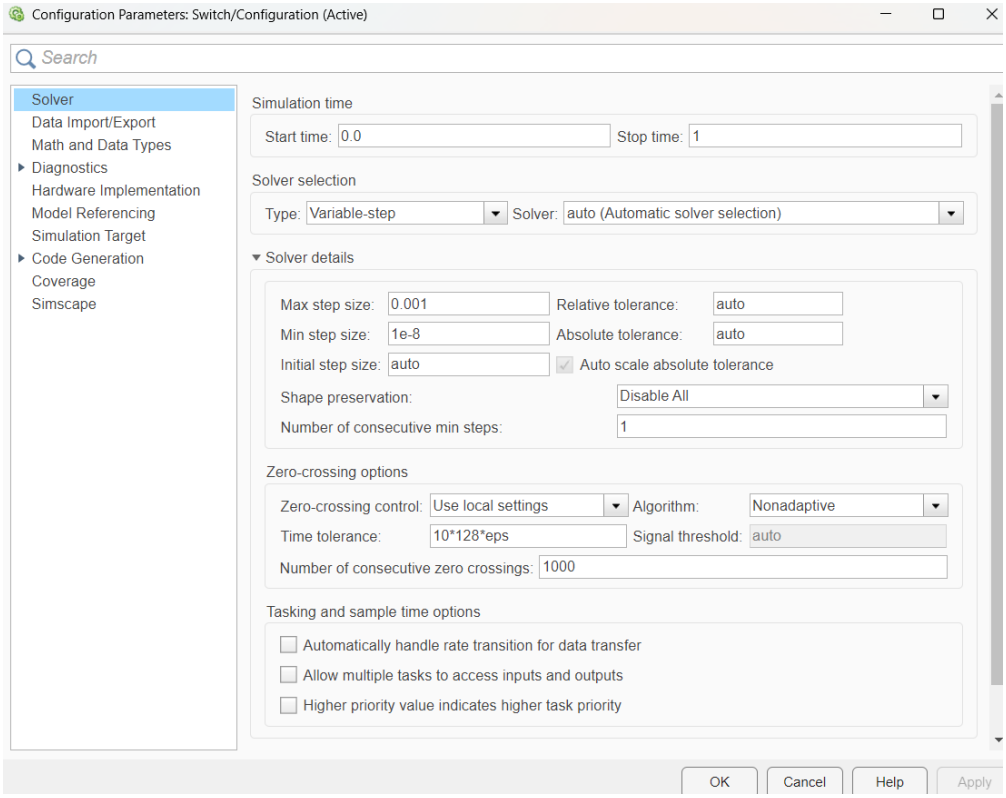


Figure 7: Solver Configuration in Simulink

The maximum step size is limited to 0.001 seconds to achieve high temporal resolution, enabling precise tracking of rapid changes in the load profile, such as sudden variations in current or voltage. A minimum step size of 1e-8 fig(7) seconds ensures that fast transients are captured correctly and also prevents solver failures during unexpected switching or oscillations.

Other key settings include relative and absolute tolerances set to `auto`, which automatically control the error based on the system dynamics. Zero-crossing detection allows for the accurate detection of discrete events, such as switch operations, without significantly increasing computational cost. Shape preservation is disabled, allowing the solver to prioritize the accuracy of the solution rather than constraining the trajectory.

3.3 Controlled AC Power source with Harmonics

This simulation implements a three-phase voltage source containing harmonic distortions in addition to the fundamental component. Harmonics are critical in evaluating the performance of power systems under non-ideal conditions, such as the presence of nonlinear loads or switching devices.8[15]

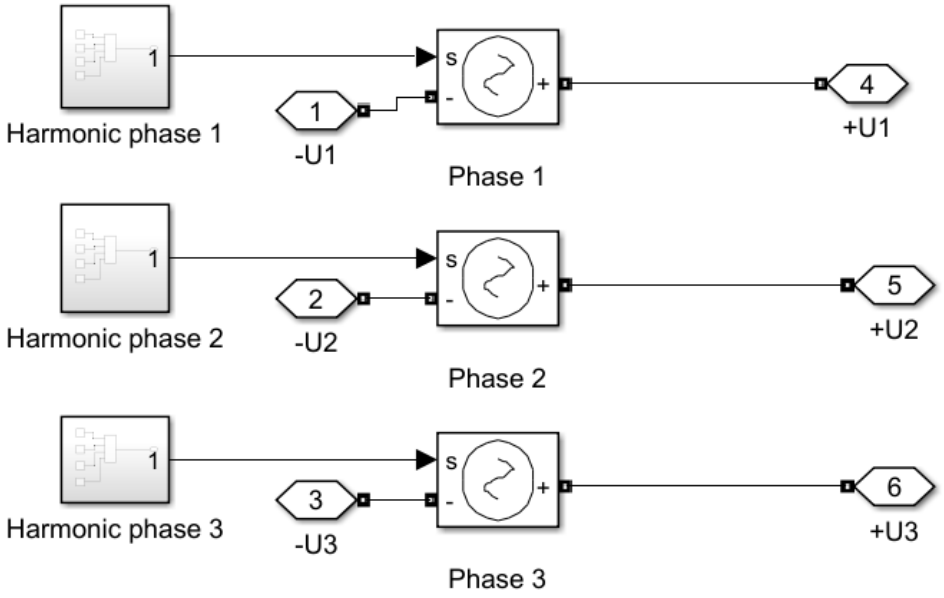


Figure 8: 3-phase Controlled AC power source in Simulink

The goal is to model and synthesize a three-phase voltage waveform that includes the fundamental frequency (50 Hz) along with the 3rd, 5th, and 7th harmonics. These harmonics are modeled based on typical amplitude percentages observed in practical systems and mostly above mentioned harmonics components are presented in system and cause of damage of electrical system so including harmonics will helps to understand and practical scenario of Industrial as well as distribution system.

Each phase is constructed by summing sinusoidal signals using Simulink sine wave blocks. The generated waveform for each harmonic is described by the equation:

$$V(t) = A \cdot \sin(2\pi f t + \phi)$$

where:

- A: Amplitude of the waveform
- f: Frequency in Hz
- ϕ: Phase angle (set to 0 in this simulation)
- t: Time in seconds

Harmonics: A harmonic is a wave or signal whose frequency is an integral (whole number) multiple of the frequency of the same reference signal or wave.The fundamental frequency or original wave is known as the first, or 1st,, harmonic. The following harmonics are called higher harmonics. The fundamental frequency of all harmonics is periodic, and the total number of harmonics is also periodic at that specific frequency.(i.e. For 3rd order component frequency will be triple than fundamental component so if in our system fundamental component is 50hz that means 3rd harmonic component will be 150hz frequency with certain amplitude.) [16]

Amplitude Reference: The fundamental voltage amplitude is set to:

$$V_1 = 230 \times \sqrt{2} \approx 325.27 \text{ V}$$

This serves as the reference for calculating the amplitudes of the harmonic components. 9 The following harmonics are introduced:

Table 1: Harmonic Component Configuration

Harmonic Order	Amplitude	Frequency (rad/s)	Amplitude (% of 1st)
Fundamental (1st)	$230 \cdot \sqrt{2}$	$2\pi \cdot 50$	100%
3rd Harmonic	$(230 \cdot \sqrt{2}) \cdot 0.2$	$2\pi \cdot 150$	20%
5th Harmonic	$(230 \cdot \sqrt{2}) \cdot 0.05$	$2\pi \cdot 250$	5%
7th Harmonic	$(230 \cdot \sqrt{2}) \cdot 0.01$	$2\pi \cdot 350$	1%

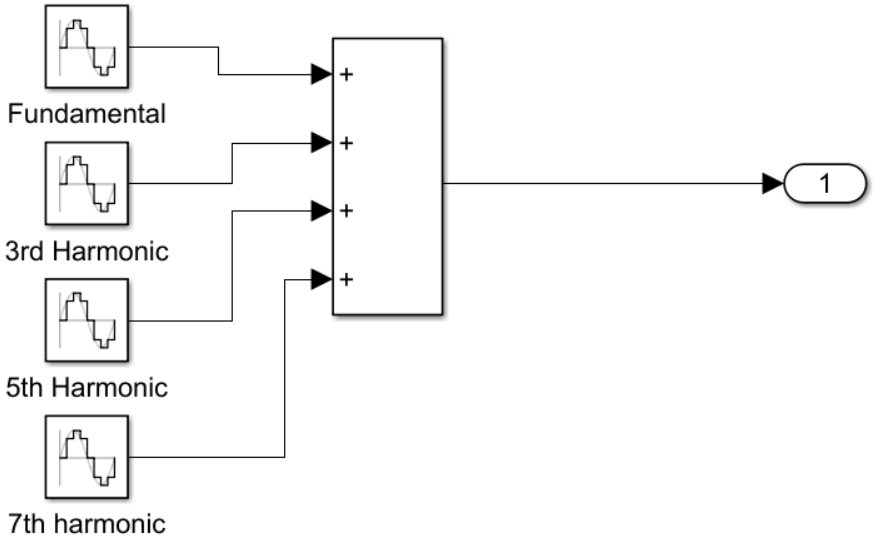


Figure 9: Harmonics Signal for Voltage source

Typical Harmonic Ranges: 9 These selected amplitudes commonly used engineering assumptions for harmonic modeling and simulation. IEEE 519-2014 provides the most authoritative harmonic limits but does not prescribe specific internal harmonic component

ratios. According to IEEE 519-2014 typical harmonic levels in industrial or commercial environments are shown below:

Table 2: Typical Harmonic Amplitude Ranges

Harmonic Order	**Typical Amplitude Range (% of Fundamental)**
3rd Harmonic	10% – 20%
5th Harmonic	5% – 15%
7th Harmonic	3% – 10%

Three-Phase System Assembly: Each phase (Phase U, V, and W) uses the same harmonic structure but with respective phase shifts (e.g., 120° separation in a balanced system). The signals from harmonic blocks are summed and connected to the three-phase source block as shown in the simulation diagram.9 After that Harmonics block employed on to each phase block according to desired harmonics Components in each phase. On simulated load profile in all three phase all three harmonics component was presented.

Voltage with harmonic components: 10 As from image 9 the plot shows multiple waveforms, and it can be noticeable which ones represent the ideal three-phase voltages and which ones include harmonics:

Smooth Sinusoidal Waveforms (light Blue, Green and light Purple): These waveforms represents pure sinusoidal wave. They represent the fundamental frequency components of the three-phase voltages, which is what could be ideally seen in a system without significant harmonic distortion. Notice how each color-coded waveform reaches its peak at a different point in time, separated by the characteristic 120-degree phase shift.

Distorted Waveforms (Red, Purple and Blue): These waveforms deviate from a perfect sinusoidal shape. They show additional fluctuations and irregularities superimposed on the fundamental frequency. These distortions are indicative of the presence of harmonics.

3.4 Power Measurement and Calculation

This section details the methodology employed for measuring power within the MATLAB Simulink model. The model utilizes a combination of voltage and current measurement units, along with dedicated power measurement blocks, to accurately determine the electrical characteristics of the system under varying conditions.

3.4.1 Voltage Measurement

Voltage measurements are performed using the Voltage Measurement block. This block is crucial for acquiring the voltage signals at specific points within the circuit, enabling the calculation of power and other relevant electrical quantities. As described in the MathWorks documentation, the Voltage Measurement block provides accurate voltage readings, which serve as a fundamental input for subsequent power calculations. The block is configured to measure the Single-phase voltages.[17]

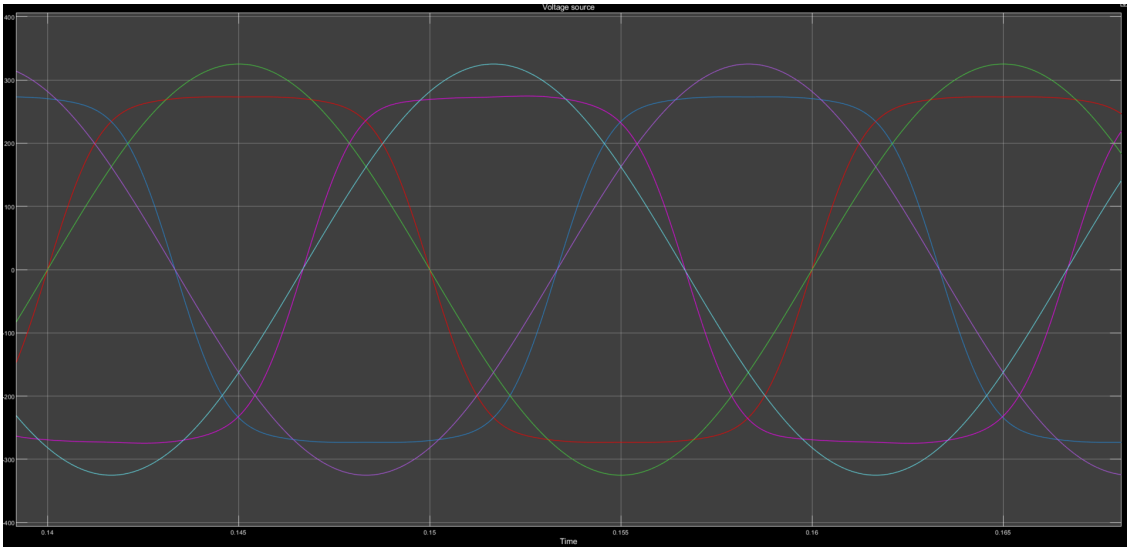


Figure 10: Three phase Voltage with and without Harmonics

3.4.2 Current Measurement

Current measurements are performed using Current Measurement blocks. These blocks are essential for obtaining the current signals flowing through different parts of the circuit. Accurate current measurements, in conjunction with voltage measurements, are necessary for calculating active and reactive power.[18]

3.4.3 Active and Reactive Power Measurement

Active and reactive power are calculated using the Power Measurement block. This block takes the measured voltage and current signals as inputs and computes the active (P) and reactive (Q) power components. The Power Measurement block, as detailed in the MathWorks documentation, provides accurate and reliable power measurements, enabling a detailed analysis of the system's power flow and efficiency. The block calculates instantaneous power, which is then used to derive the active and reactive power components. The power factor is also calculated.

The **Power Measurement block** [19] is designed to evaluate the real and reactive power associated with a single-phase voltage and current waveform, including harmonics. It is applicable to both sinusoidal and nonsinusoidal periodic signals. This block is particularly useful for frequency-domain analysis, enabling insight into power distribution across various harmonic components.

Overview of Parameters

- **Sample Time**: Defines whether the block operates in continuous (0) or discrete-time (non-zero value) mode.
- **Harmonic Numbers**: A vector specifying which frequency components should be included in the analysis. 11In filed of Harmonic number user should enter the component

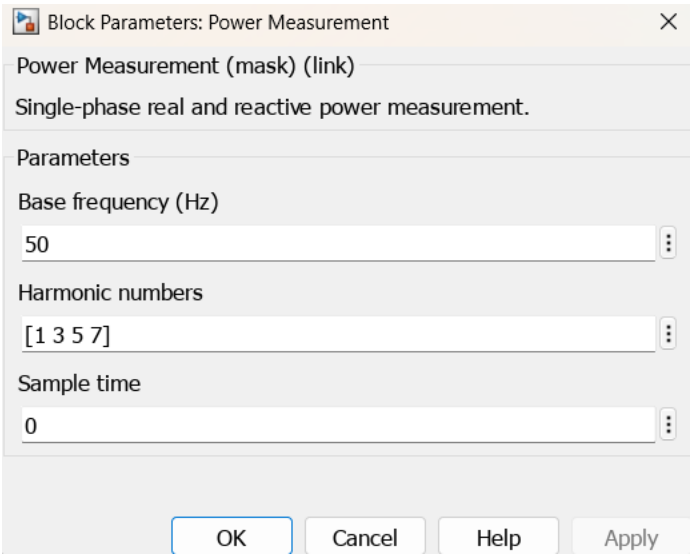


Figure 11: Configuration of Power Measurement block

must be ana lyse so as in present model there are 3rd, 5th and 7th harmonic components were there. So that can be analyses by entering in field of Harmonic number [1 3 5 7]

- 0 for DC component, 1 for the fundamental frequency,
- $n > 1$ for higher-order harmonics.

Mathematical Foundation The real and reactive powers for each specified harmonic component k are derived using phasor-based analysis. A phasor represents a sinusoidal signal in terms of its magnitude and phase, facilitating power computation in the frequency domain.

The complex power for harmonic k is given by:

$$P_k + jQ_k = G \cdot \left(V_k e^{j\theta_{V_k}}\right) \cdot \left(I_k e^{j\theta_{I_k}}\right)^*$$

Where:

- P_k: Real power of the k^{th} harmonic.
- Q_k: Reactive power of the k^{th} harmonic.
- G: A scaling factor:
  - $G = 0.25$ for DC component ($k = 0$),
  - $G = 0.5$ for AC harmonics ($k > 0$).
- $V_k e^{j\theta_{V_k}}$: Phasor of voltage at harmonic k.
- $I_k e^{j\theta_{I_k}}$: Phasor of current at harmonic k.

- $\left(I_k e^{j\theta_{I_k}}\right)^*$: Complex conjugate of the current phasor.

Phasor Extraction To compute $V_k e^{j\theta_{V_k}}$ and $I_k e^{j\theta_{I_k}}$, the block uses Fourier-based integration over one signal period T, where $T = \frac{1}{F}$ and F is the base frequency:

$$V_k e^{j\theta_{V_k}} = \frac{2}{T}\int_{t-T}^{t} V(t)\cdot\sin(2\pi kFt)\,dt + j\cdot\frac{2}{T}\int_{t-T}^{t} V(t)\cdot\cos(2\pi kFt)\,dt$$

$$I_k e^{j\theta_{I_k}} = \frac{2}{T}\int_{t-T}^{t} I(t)\cdot\sin(2\pi kFt)\,dt + j\cdot\frac{2}{T}\int_{t-T}^{t} I(t)\cdot\cos(2\pi kFt)\,dt$$

These integrals effectively project the signal onto sinusoidal basis functions, isolating the contribution of each harmonic frequency.

Total Power Calculation If the user specifies harmonics $k = 0, 1, \ldots, n$, then the total power values are obtained by summing over these components:

- **Total Real Power**:
$$P = \sum_{k=0}^{n} P_k$$
- **Total Reactive Power (excluding DC)**:
$$Q = \sum_{k=1}^{n} Q_k$$

Note: The DC component ($k = 0$) does not contribute to reactive power it only contributes to real power.

3.4.4 Apparent power and Power Factor

Figure 12 illustrates the Simulink model implemented for real-time power measurement and power factor calculation of an electrical system. The model takes instantaneous voltage (V) and current (I) signals as inputs, sourced from input blocks labeled `1` and `2`, respectively. These signals are fed into a `Power Measurement` block configured for continuous-time analysis.

The `Power Measurement` block calculates the active power (P), reactive power (Q), and the RMS values of the voltage (U_{rms}) and current (I_{rms}). The active power output is directly labeled as `out.P1` and described as "Active Power". Similarly, the reactive power output is labeled `out.Q1` and described as "Reactive Power". To compute the apparent power (S)[20] , the model squares the active power (P) and reactive power (Q) signals using `u^2` blocks. These squared values are then summed using an `Add` block. The square root of this sum, calculated using a `Sqrt` block labeled `u`, represents the apparent power (S), which

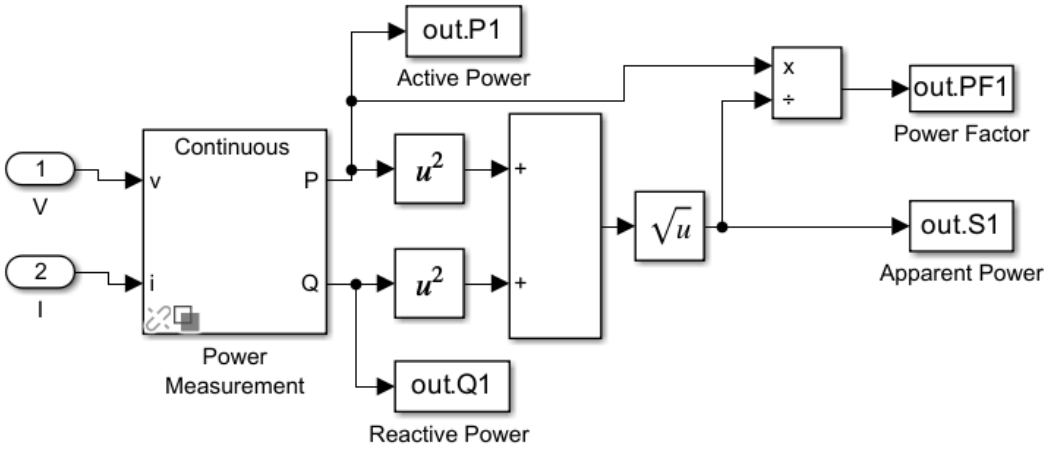


Figure 12: Power Measurement unit on each phase

is outputted through the `out.S1` block labeled "Apparent Power". Mathematically, this is represented as:

$$S = \sqrt{P^2 + Q^2} \tag{1}$$

Finally, the power factor (PF)[20] is determined by dividing the active power (P) by the apparent power (S). This division is performed by a `Divide` block, with the active power connected to the numerator and the apparent power connected to the denominator. The resulting power factor is outputted through the `out.PF1` block labeled "Power Factor". The power factor is thus defined as:

$$\text{PF} = \frac{P}{S} = \frac{P}{\sqrt{P^2 + Q^2}} \tag{2}$$

The continuous-time nature of the `Power Measurement` block allows for dynamic analysis of power quantities and power factor fluctuations in the system.

3.5 Configuration For Data Recording

For the purpose of visualizing energy consumption and recording data for load forecasting, the `To Workspace`[21] block in Simulink serves to write simulation data to the MATLAB workspace for further analysis or processing. In this specific configuration, the block is set to save the input signal to a variable named `V1ind`.

For each phase, the recorded data includes voltage, current, active power, reactive power, apparent power, and the power factor. All data points generated during the simulation are captured, as shown in fig 13 indicated by the `inf` setting for the data point limit. The decimation factor is set to 1, meaning that every data point is saved, and the data is stored as a MATLAB `timeseries` object, which is suitable for signals with a time component.

Furthermore, if the input signal contains fixed-point data, it will be logged as a `fi` object to preserve its properties. The sample time is configured to be 0.001 seconds, implying that data will be recorded with a sampling rate of 1000 Hz which can be vary as per requirement.

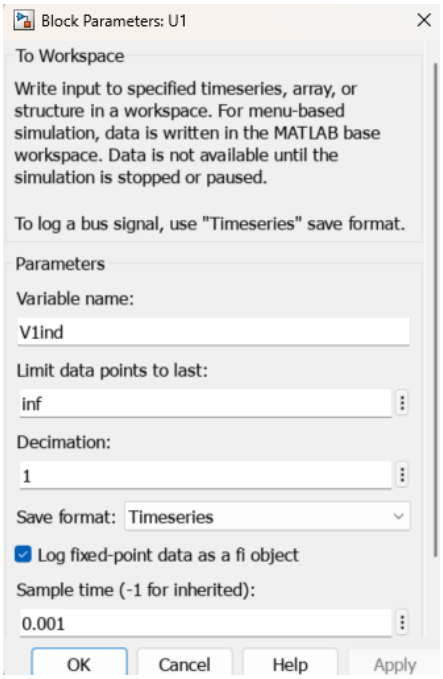


Figure 13: To workspace block Configuration

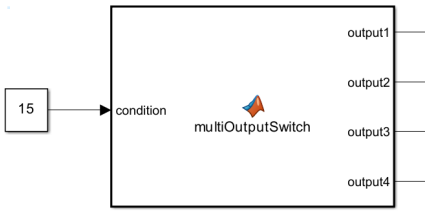


Figure 14: Switch

3.6 Load Selection

To enable user-defined load selection in the simulation model, a dynamic switching mechanism was implemented using a custom MATLAB function block, referred to as the `multiOutputSwitch` 14[22] . This block is integrated with a GUI, where users can select one or more loads to observe their energy consumption profiles.

Upon selection, the GUI sends a constant numeric value representing the selected load combination to a Python interface. The Python script, acting as a middleware, transfers this value to a MATLAB script, which passes it as an input to the `multiOutputSwitch` function block.

The `multiOutputSwitch` receives this numeric code (ranging from 1 to 15) as a condition input and activates the corresponding outputs to control ideal switches connected to individual loads. Each output from the switch block corresponds to one of the four loads in the model. The function supports all possible combinations of four loads (single, multiple, or all loads simultaneously), as defined by 15 conditional cases.

This mechanism allows the model to selectively activate only the loads chosen by the user, thereby enabling focused energy consumption analysis and visualization for each load or combination thereof. The switching logic ensures that gate signals are sent only to the

desired load blocks, as seen in the Simulink diagram (refer to Figure 6). *Note: The MATLAB function code implementing the condition logic is attached with this report for reference.*

3.7 Replicated Load Profiles

To create simulation that are very closely resemble real-world industrial energy behavior, it is necessary to replicate realistic load profiles. This is goes beyond using simple generic signals—accurate modeling demands a very technical understanding of how various electrical loads behave over time.

Importance of Technical Knowledge in Simulation

Simulations without the technical knowledge often rely on the idealized or generic waveforms that fail to represent the complexities of the real industrial loads. In contrast, simulations informed by real engineering insights are capture realistic load behaviors. For example, instead of applying a simple sinusoidal signal to a general load, a technically accurate setup would involve using a controlled AC power source to drive an induction motor, resulting in a load profile that mirrors real operational dynamics.

This technical realism allows:

- **Precise classification of load types**, such as resistive, inductive, or motor-based.
- **Accurate modeling of time-based consumption**, including startup surges and cyclical variations.
- **Trustworthy interpretation of simulation results**, especially when diagnosing inefficiencies, harmonics, or power quality issues.

Justification for Load Types Used in Simulation

A variety of common industrial loads were considered when developing the replicated profiles:

- **Induction Motors**: These are the workhorses of industry, used in pumps, compressors, and machine tools. They are robust, low-maintenance, and compatible with variable frequency drives (VFDs) for speed control.[23]
- **Induction Furnaces**: High-power, non-linear loads used for melting or heating metals. They provide clean, localized heating but introduce harmonic distortion and dynamic power demand.[24]
- **Resistive Loads**: Found in lighting systems, heaters, and test loads, they are linear and predictable—ideal for baseline and thermal energy profiling.

Relevance to Cyber-Physical Energy Management

In modern Industry 4.0 systems, energy management relies on accurately simulating the interaction between digital control systems and physical electrical loads. The replicated load

profiles contribute by:

- Reflecting the **mix of linear and non-linear loads** typical in industrial environments.
- Allowing simulations to reproduce **realistic power demands and usage patterns**.
- Helping detect and resolve issues such as **inefficiencies, voltage drops, harmonic distortions**, and energy waste.
- Preparing systems for **real-world challenges**, such as unexpected load spikes or the addition of new machinery.

By replicating true industrial load behaviors within simulation environments, this approach improves both the fidelity and usefulness of the simulated data—ultimately enabling smarter, more responsive energy management systems.

3.8 Load Cycle Patterns

Generally, the load of industrial consumers follows certain patterns over the day and the week cycle, that highly depend on the operation times. [2] derived a curve for the energy consumption over a typical work day that mostly follows a one-shift pattern from morning to evening with a little dip during lunchtime. Also, the energy demand flattens slowly in the afternoon indicating some operations are still active even if main consumers have switched off. Even during the night, a certain base load is kept. The curve obtained by Tseroukas et. al. is shown in 15. In the following, different ways to model such a daily or weekly load

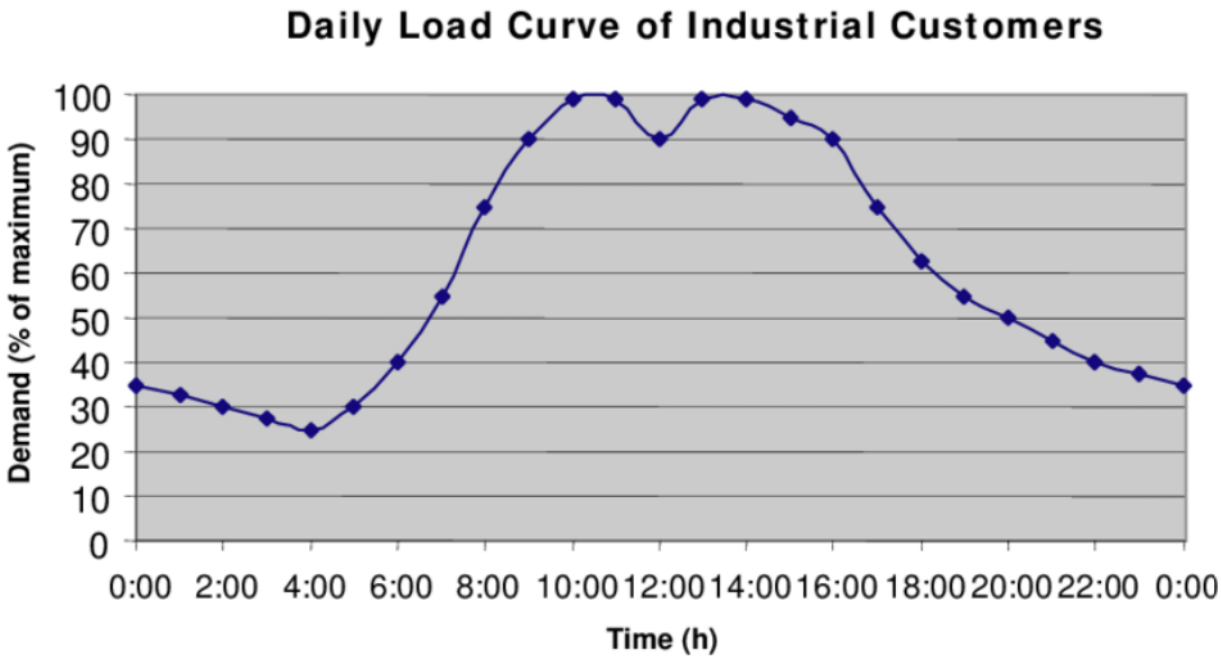


Figure 15: Typical load curve of an industrial consumer over the day according to Tsekouras et. al. [2]

curve will be discussed. One way is to model the curve as a polynomial function. In order to do so, the values proposed by Tseroukas et. al. are slightly modified in the way that the base load is set to a constant between 4am and 8pm; a value of 20 percent is taken as base. For this purpose, the load decrease in the afternoon and evening hours is modeled steeper to get down to the base load sooner. The daily demand curve is fitted to a polynom in Python giving hourly percentage values, rearranged around the noon between minus and plus eight. A polynom of degree nine is selected for modeling the daily demand curve, which is shown in 16. Later on, this normalized curve could be scaled down to achieve values that match 80 percent of the maximum load, so excluding the 20 percent base load. Including a

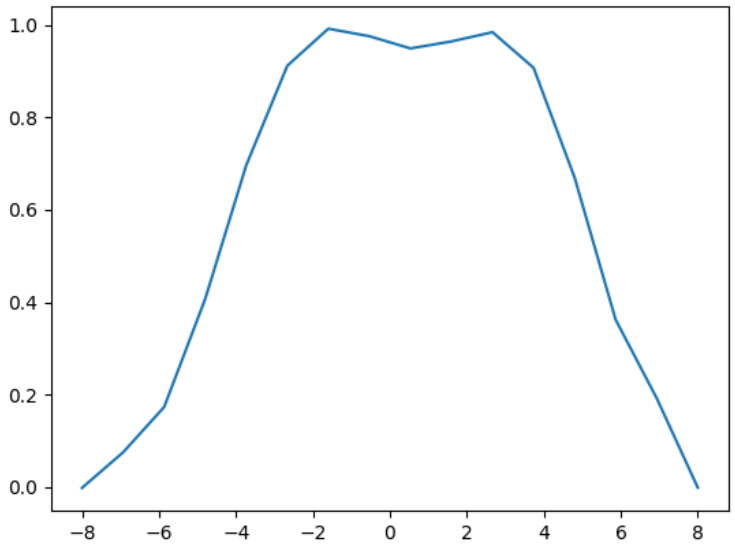


Figure 16: Polynom of degree 9 fitted to the modified industrial demand curve from Tsekouras et. al. [2]

typical weekly cycle consisting of five work days and two weekend days, the constant base load of the night can be continued over the weekend days. The week cycle constructed by Python therefore looks as shown in 17. Therein, the normalized factors are applied to a given maximum load, exemplary 1000 kW in this case, with a granularity of one minute. To further modify the literature values to reflect little deviations, an additional random factor is added every minute in order to simulate a rather non smooth behavior. The random factor is up to plus-minus ten percent of the maximum load within the daily range and up to plus-minus five percent of the maximum load for the base periods. An exemplary result is shown in figure 17

Besides the usual weekly cycle, also holidays can impact the load, by means they are usually days without operations such as the weekend is. Since holidays appear acyclic to the weekly cycle based on the calender date or the moon cycles, including such in the forecast would require the integration of a calender: By labeling every holiday as a non-work day based on the national schedule. Also, the schedule can be modified to reflect planned out-times of the

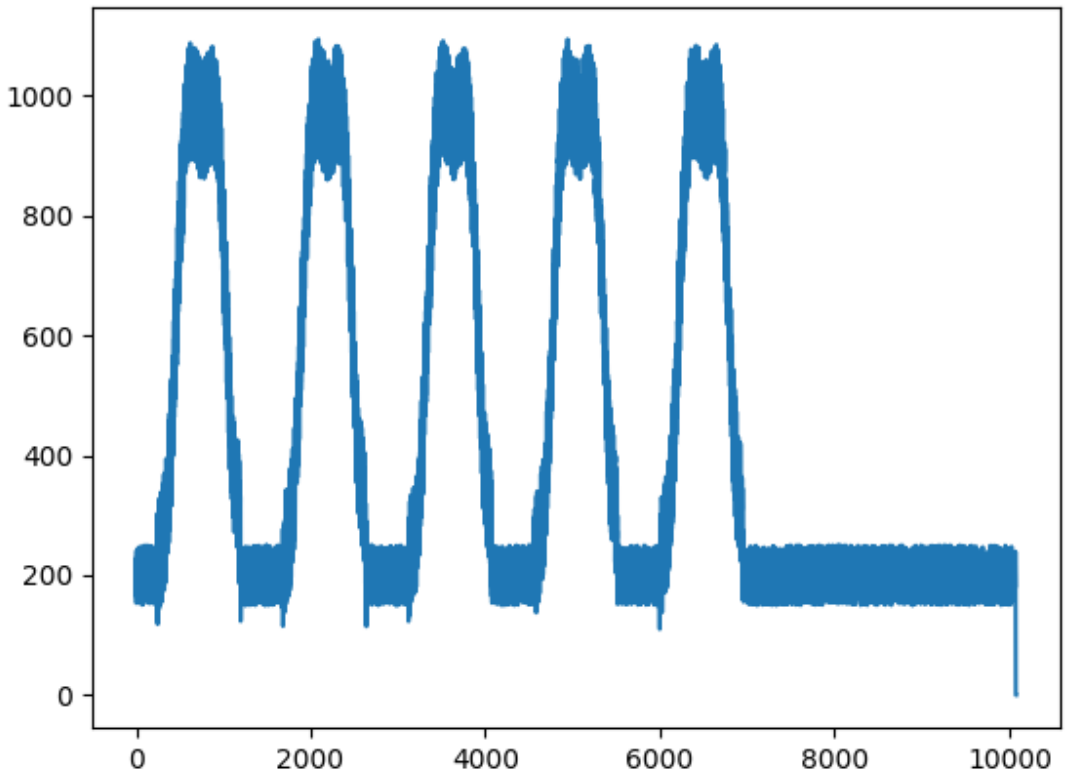


Figure 17: Model of an exemplary weekly load cycle for a 1000 kW maximum consumption including random deviations based on the modified daily curve by [2]

plant, such as free days or shutdowns.

The above provides an overview on considerable factors when forecasting the load cycle of a plant. Besides doing the polynom fit in Python, also Matlab function blocks can be added to a Simulink model to ramp up or down or randomize a normalized multiplication factor for simulation branches. However, the method used for this model is based on step functions that switch on or off at a certain time in the simulation, which is described in the next chapter. Whereas the polynomial fitting approach above rather represents the average value of a factory with many consumers, the simulation in this application is yet representing a rather small factory with only a handful of machines. Therefore, a direct on- and off switching better represents the given simulation. Notable, that this direct switching would result in a more smooth ramp up if it was passed through a low-pass filter.

3.9 Synthetic Data Generation for Load forecasting

In practical applications where the real-time measurement data is an either unavailable, incomplete, or d to capture for extended timeframes, **synthetic data** serves as a reliable alternative to replicate the real-world electrical behavior. These approach is specially useful in energy modeling and load forecasting, where understanding daily demand fluctuations is critical for planning and optimization.

Time Compression Model

To simulate an entire day (24 hours) within a manageable simulation runtime, a **time-compression approach** was implemented:

- 24 seconds of simulation time represent 24 real-world hours.

Thess scaling allows faster computation while preserving time-dependent patterns of energy consumption.

Load Scheduling According to Demand Curve

In this model, loads is not applied uniformly across the day. Instead, they are introduced based on a time-based demand curve that reflects how industrial facilities typically operate:

- A **base load** remains connected to the power source throughout the entire 24-second simulation. This represents always on systems such as lighting, server rooms, or baseline HVAC systems.
- Additional loads are scheduled during working hours:
  - From 06:00 to 18:00
  - From 08:00 to 16:00
  - From 10:00 to 14:00
- **Peak and break periods** are simulated:

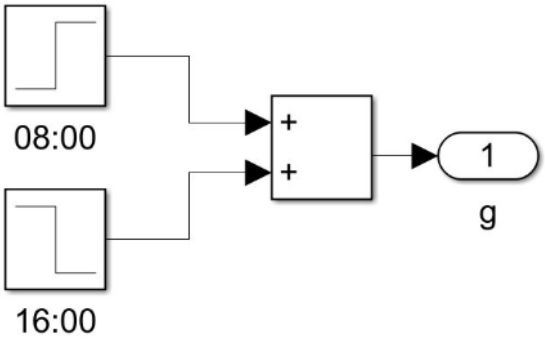
– 11:00 to 12:00 (peak operational demand)

– 13:00 to 14:00 (post-break restart or shift change)

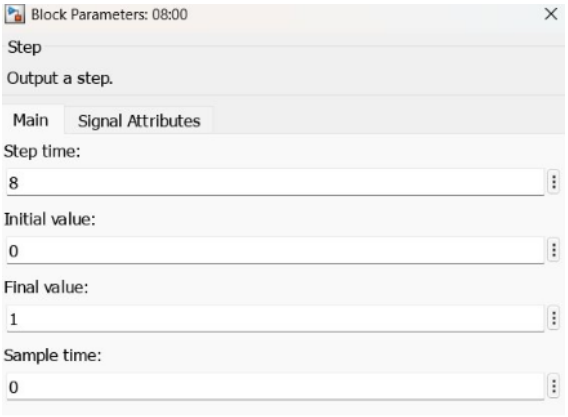
These time slots replicate staggered machine usage, shift-based operation, and real consumption patterns found in the industrial facilities.

Use of Step Functions for Load Control

The entire load scheduling mechanism is built using **step functions**, a standard technique in control system modeling. Each step function acts as a digital gate signal, activating or deactivating a load at a specific time.[25]



(a) Step block simulating load at 08:00



(b) Step block parameters window

Figure 18: Simulink step functions used for scheduling synthetic load profiles

- A step time of `8` seconds in the simulation represents 08:00 hours.
- `Initial value` is set to 0 (load is off).
- `Final value` is set to 1 (load is activated).

Multiple step functions can be added together to create a combined control signal, often denoted as g, that governs load switching logic.

Importance in Load Forecasting

Synthetic load generation provides us a structured and flexible test environment for forecasting systems:

- Enables **training of machine learning models** with labeled and predictable data.
- Allows **validation and fine-tuning of forecasting algorithms**.
- Supports **scenario testing** with different operational schedules.
- Facilitates **stress testing** under simulated peak demands.

By emulating real-world operational patterns through synthetic signals, this approach forms a robust foundation for developing and evaluating load forecasting strategies in smart grids and industrial energy systems.

4 System Architecture: Simulation Execution and Data Handling

4.1 Acquired Software and Data Flow

The whole project is realized either with free program versions or with available student or university licenses contained in the stack of available software of the UAS Cologne. The architecture of the system is designed to simulate energy profiles under various load conditions and present real-time results via a user-friendly interface. This involves a tightly coupled integration between MATLAB and Python, a time-series database for efficient data storage, and a responsive front-end built using Streamlit.

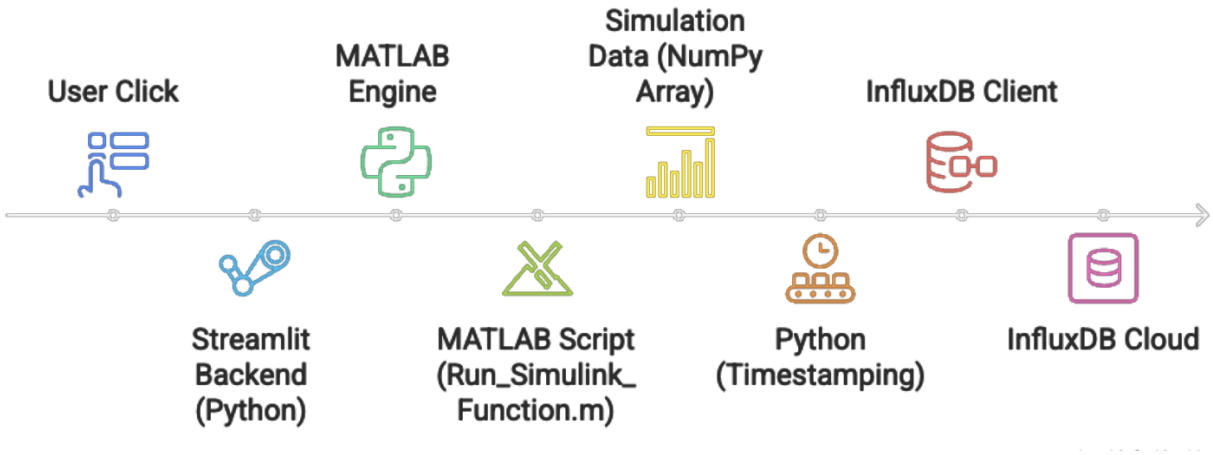


Figure 19: End-to-end system flow diagram showing the integration between the Streamlit interface, MATLAB-based simulation engine, Python preprocessing, and Influxdb for time-series data storage and visualisation.

The main function will run in Python, which is selected because of its very high connectivity to the other programs. Also, Python has a wide spread application base in data science which comes with a large stack of usable functions, such as forecasting algorithms. Python is also well documented and freely available. Eventually, the connectivity was the argument beating out earlier considerations to use Matlab as the main function code.

However, the capabilities of Matlab and Simulink for machine modeling are undoubted. Based on the author's experience, it is often easier to find supporting documentations and tutorials for Matlab software than for other alternatives such as Open Modelica. Therefore, Simulink is used herein to generate the synthesized data. Such requires a connection between Python an Simulink, which is implemented via a Matlab Function; the Matlab engine is started by Python, which then also call the Matlab function that will trigger the start of the Simulink simulation. In the reverse way, the simulation results will be passed on to the Matlab function where they are slightly formatted and given to the Python program. The main program will calculate averages and writes the complete as well as the aggregated data

into a database. InfluxDB is chosen as a database because of it's easy to use interface and compatibility with Python. As well, Influx DB offers a free, limited version that can be used for the implementation of this project in low scale. However, this free version comes with a 30 day auto-deletion period. Future work therefore might consider setting up an own database server or looking for other alternatives. Clearly, using InfluxDB for a broader, long-term monitoring would benefit from a subscription model. The user interface (HMI) is implemented via the Python main code in Streamlit. Such bears the benefit of direct and active user interaction, meaning that the user cannot only see existing data but can also make selections and restart the simulation. Therefore, the final decision is to use Streamlit over Grafana, which has been considered earlier because of its time series capabilities. Grafana plugins allow slight direct user interaction, usually coming from the business intelligence domain, such as the one from Volkov Labs. [26]

The GRU model runs separately, taking in the historic synthesized data from the simulation output as a .csv file. Overall, there are three Python files, one main file and one for each of the two HMI pages. There is one .m file with the Matlab function in text-code format and one .slx file with the Simulink model. The graphics 20 shows an overview of the infrastructure components used and the data flow. Over the whole development, a certain degree

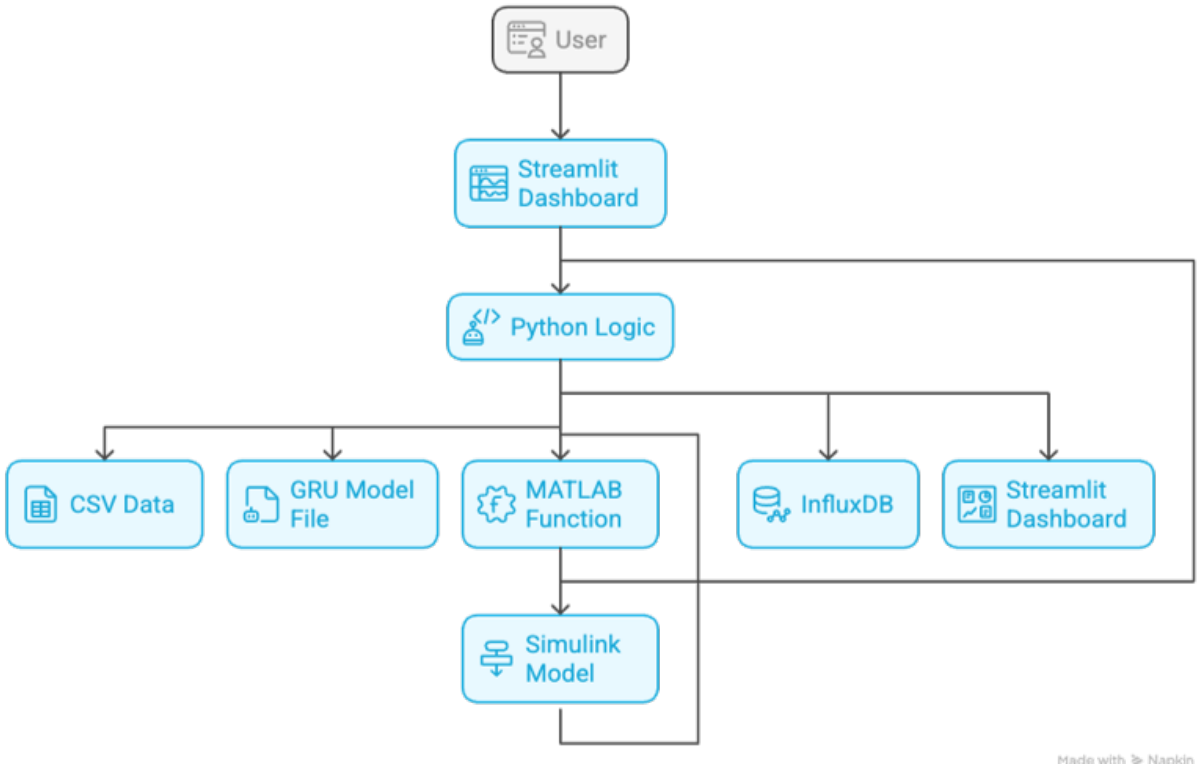


Figure 20: Data Flow between the Selected Program Infrastructure Elements of the Application (created with Napkin.ai)

of modularity is intended, in order to make the program extendable, especially with more different machine types. In this context, also scalability can be considered. Also, the code should rely on commonly available Python libraries.

4.2 HMI: Homepage

The main interface of the Cyber Physical Energy Management System (CPEMS) is developed using Streamlit [27], designed for user-friendly navigation across different modules. Upon launching the application, users are welcomed with a clean, wide-layout home screen featuring two primary interaction boxes: Forecasting of loading and industrial profile.

- Load Forecasting Module: Directs the user to a page where they can train the GRU-based forecasting model and generate energy predictions across multiple horizons.
- Industrial Profile Module: Provides access to MATLAB-Simulink simulations and displays simulation data stored in Influxdb via visual dashboards.

Each module is represented by a clickable visual box, styled using HTML and CSS within Streamlit markdown blocks. The interface ensures intuitive access by maintaining a persistent sidebar with navigation options. The layout is optimised using columns and flexbox styling for responsive, organised content alignment.

The code structure separates navigation logic and UI components, allowing scalable and modular expansion of the system. The full implementation is provided in the appendix.

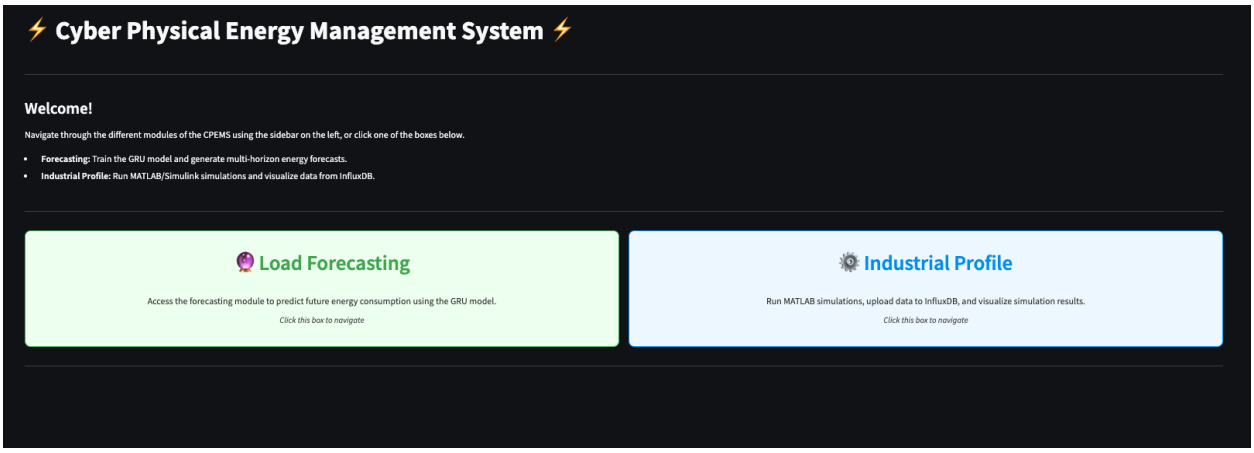


Figure 21: CPEMS HMI Main Page – The user-friendly home screen provides direct access to forecasting and industrial simulation modules, supported by a clean layout and persistent sidebar navigation.

4.3 Integration and Preprocessing in Python

The simulation process begins when the user selects a specific load combination and initiates the simulation from the Streamlit interface. This action triggers a Python backend function that interfaces with MATLAB using the MATLAB Engine API [28]. The selected load configuration is passed from Python to a MATLAB function `run_simulation(load_combination)`, which performs the simulation and returns the results as MATLAB variables.

The simulation generates a comprehensive set of electrical parameters for each time step. This includes voltages (U_1, U_2, U_3), currents (I_1, I_2, I_3), active power (P_1, P_2, P_3), reactive

power (Q_1, Q_2, Q_3), apparent power (S_1, S_2, S_3), and power factor (PF_1, PF_2, PF_3). These are returned as arrays and mapped using format-specific dictionaries.

Two distinct formats are supported for this data:

- **Siemens Format:** Provides a full set of 18 parameters including voltages, currents, active/reactive/apparent power, and power factor.
- **Beckhoff Format:** A simpler format containing only voltages and currents (U_1 to U_3, I_1 to I_3).

A field mapping dictionary is used in Python to decode and standardize the parameter indices based on the format. The resulting arrays are converted into a structured `pandas` DataFrame that includes a timestamp column and columns for each electrical quantity. Any necessary preprocessing (e.g., column renaming, filtering missing fields) is performed to ensure consistency before storage.

4.4 Matlab Function Code

The Python file will first start a Matlab engine. By doing so, the Matlab program session must not be opened manually by the user. Following up, Python calls a Matlab function in .m code format. The function call includes two variables that define the model name and the machine types to be switched "ON" in the simulation. To enable the machine type selection, the Simulink program contains a switch block, that will enable or disable certain branches based on the numerical input value. The Matlab function will start the Simulink session and open the simulation file automatically once being called. Matlab will write the simulation outputs into a matrix that contains the timestamp and for each of the three phases the voltage, the current as well as the active, reactive and apparent power and the power factor. The matrix with rough data is given as the function output, that is transferred to Python, where it is written into a Numpy array. Python finally closes the Matlab engine. The described interaction is shown in figure 22 with selected code snippets: It is displayed, how the two Python arguments for the model name and the condition value turn into Matlab function arguments b and c. The output of the Matlab function, the matrix a, is passed on to Python as `raw output` written into a Numpy array.

4.5 Database

Once the preprocessing is complete, the DataFrame is written into an Influxdb database[29]. Influxdb is selected for its time-series optimized architecture, enabling high-speed writes and efficient queries. Each data point includes a timestamp, all measured quantities (voltages, currents, powers, power factors), and metadata such as the `load_combination` and the data `format` (Siemens or Beckhoff).

This organised structure allows for both detailed historical analysis and rapid retrieval of specific metrics for visualisation. The database supports persistent storage, enabling the user interface to access past simulations without rerunning them.

```
"""Start the MATLAB engine."""
self.eng = matlab.engine.start_matlab()
self.eng.cd(self.matlab_script_path, nargout=0)
```

```
# Pass the correct arguments to MATLAB function
raw_output = self.eng.Run_Simulink_Function(model_name, condition_value, nargout=1)
```

```
function [a] = Run_Simulink_Function(b, c)
    mdl = b;
    x = c;
    % Open and configure the Simulink model
    open_system(mdl);
```

```
% Run the simulation
out = sim(simIn);
```

```
% Write the matrix to the output variable
a = OutMatrix;
```

```
self.function_output = np.array(raw_output)
```

```
self.eng.quit()
st.info("MATLAB engine stopped.")
```

Figure 22: Code interaction between Python and Matlab)

4.6 HMI: Simulation Page

The Human-Machine Interface (HMI) is implemented using Streamlit and serves as the primary access point for initiating simulations and visualising results. On the simulation page, users can select predefined load combinations and data formats from dropdown menus and trigger the simulation process. The interface provides real-time feedback, including a progress animation during execution.

After the simulation is complete and data is stored in the database, the interface automatically queries Influxdb to retrieve the latest results. These are plotted using interactive line charts for each parameter (e.g., voltage, current, power), enabling users to analyse trends across all phases. Additionally, the results can be downloaded as a CSV file for offline analysis.

This modular architecture ensures flexibility, traceability, and ease of use. It encapsulates the complexity of simulation and data handling behind a seamless interface, providing a powerful tool for analysing power system behaviour under different configurations.

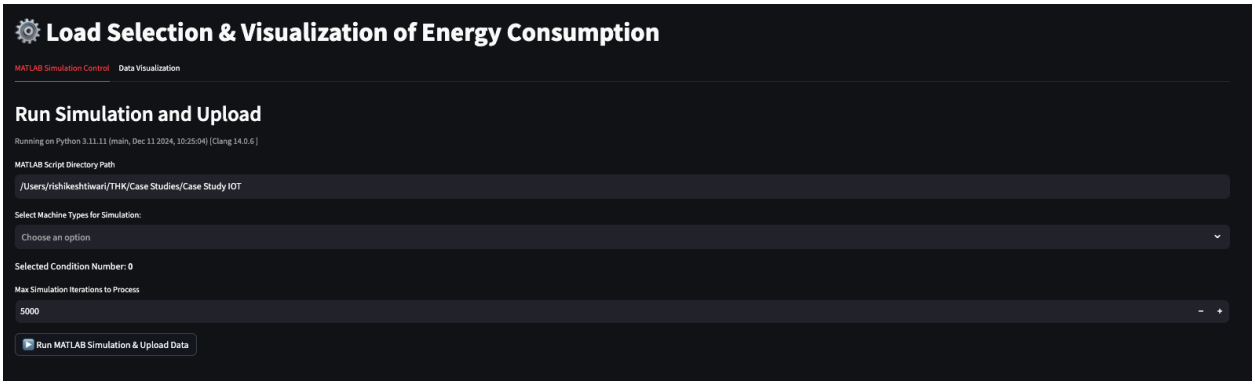


Figure 23: Streamlit-based Simulation tab interface allowing users to select load parameters, execute MATLAB simulations in real time, and trigger automatic data preprocessing and storage into Influxdb.

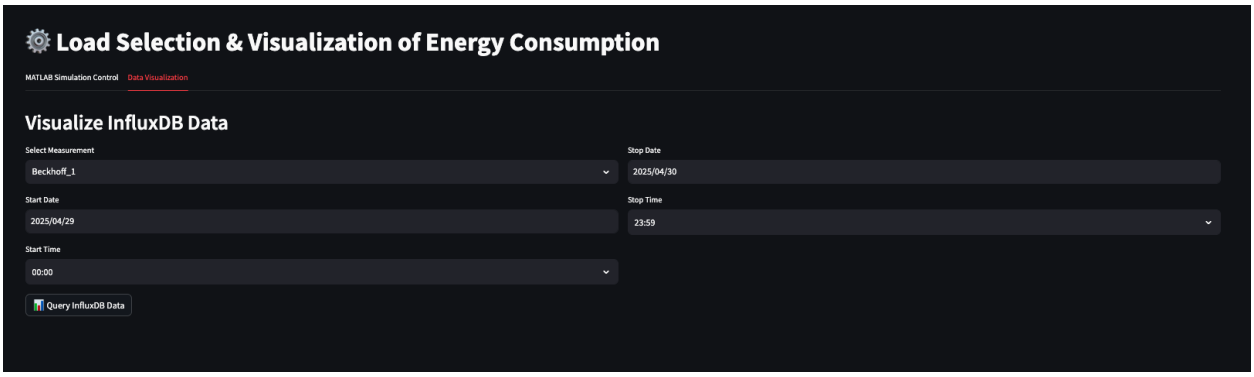


Figure 24: Visualisation tab of the Streamlit interface displaying time-series data retrieved from Influxdb, including interactive plots of voltage, current, active/reactive/apparent power, and power factor for each phase.

5 System Architecture: Load Forecasting

5.1 Fundamentals of Load Forecasting: Role of Active and Reactive Power

Load forecasting involves predicting future energy demand by analysing past energy usage patterns and considering key factors such as time, temperature, and operational behaviour. It is typically categorised into short-term and long-term forecasting. Short-term forecasting supports daily industrial operations by ensuring a stable power supply, improving energy efficiency, and enabling smoother production processes. Long-term forecasting, on the other hand, helps guide strategic planning and investment decisions, such as expanding existing facilities or building new ones to meet projected demand. Accurate load forecasting provides a range of benefits: it aligns energy supply with production needs, prevents overuse or shortages, highlights consumption patterns to identify areas for efficiency improvements, and enables smarter scheduling of high-energy tasks during off-peak hours, leading to reduced operational costs. Looking ahead, load forecasting will play a key role in managing the energy demands of industrial growth and integrating variable renewable energy sources. Ultimately, it supports the creation of a reliable, cost-effective, and sustainable energy system.[30]

As part of this forecasting effort, we have chosen to include both active and reactive power as core parameters because they provide a completer and more realistic picture of energy demand, especially in commercial and industrial environments. While active power is the portion of electricity that performs useful work, reactive power, though it doesn't directly contribute to work—is crucial for operating equipment like motors and transformers that rely on magnetic fields. Ignoring reactive power can lead to underestimating the actual load on the electrical system.[31]

By forecasting both active and reactive power, we gain a clearer understanding of total power demand, which is especially important in facilities with heavy inductive loads. This improves the accuracy of the forecast and helps manage costs, as utility companies often base charges on total power demand (measured in kVA), which includes both components. Forecasting these values enables better power factor management, helping industries reduce demand charges and improve overall efficiency.

It also provides valuable insight into power factor trends. Tracking both active and reactive power over time makes it easier to identify inefficiencies and implement corrective measures, such as power factor correction devices, to reduce unnecessary reactive demand and enhance system performance. Additionally, understanding the variations in reactive power throughout the day allows for smarter scheduling of energy-intensive operations, for instance, avoiding high reactive loads during peak hours by staggering equipment use.[32]

In summary, incorporating both active and reactive power into load forecasting results in more precise predictions, smarter energy use, reduced costs, and a more stable and efficient power system.

5.2 Model Selection: GRU

Since our forecasting system is primarily built using Python, it was an obvious choice for implementing the load forecasting model. Python is well-known as a powerful tool for data analysis, offering a wide range of time series forecasting algorithms. Its versatility and extensive library support make it an ideal choice for energy forecasting applications.

Several forecasting methods in Python have been explored in previous studies. For example, Babich et al.[33] conducted research using real energy consumption data from a company in Yekaterinburg, Russia. The dataset, collected at 30-minute intervals and free from missing values, allowed for smooth preprocessing. Their findings revealed that a support vector machine (SVM) with a radial basis function (RBF) kernel delivered the highest accuracy, followed by the ARIMA model. Other techniques tested included linear SVM and LSTM-based artificial neural networks. In another study, Arce and Macabebe from the University of Manila[34] recommended support vector regression for residential energy forecasting, especially in households with photovoltaic (PV) arrays. Their research also showed that Random Forest models performed well for next-day predictions, while linear and polynomial regression methods struggled to capture the nonlinearity of energy consumption due to PV generation. This prior research highlights the effectiveness of various forecasting models, helping us make informed decisions about the most suitable techniques for our own load forecasting tasks.

Based on our comparative analysis, the Gated Recurrent Unit (GRU) model emerged as the most suitable choice for load forecasting. GRUs offer a strong balance between performance and efficiency. They deliver accuracy on par with Long Short-Term Memory (LSTM) networks but with a simpler architecture, which results in faster training times and a lower risk of overfitting. Compared to basic Recurrent Neural Networks (RNNs), GRUs are better at capturing temporal patterns—an essential feature in load forecasting where past consumption heavily influences future demand.[35]

One of the key advantages of GRUs is their ability to handle time-series data effectively, making good use of available data and managing sequences smoothly. While models like XGBoost train quickly and are less prone to overfitting, their tree-based structure isn't well-suited for capturing the time-based dependencies inherent in sequential data. Given the need to balance accuracy, training efficiency, and the ability to model time-dependent patterns, the GRU model stands out as the most reliable and practical option for our forecasting objectives.

5.3 Forecasting Algorithm: Model Training and Forecasting Flow

A key prerequisite for load forecasting is that the historical data must contain at least 1,440 data points per day, equivalent to one data point for every minute. This level of resolution is essential for accurate model training. If the uploaded file does not meet this requirement, the system will reject the file, and the model will not proceed with training. On the other hand, if the file contains multiple data points within a single minute, the system will automatically resample the data by averaging all values within each minute. The forecasting process will then continue using this resampled dataset.

Once the historical data is uploaded, feature engineering is carried out to eliminate any irrelevant or redundant features, such as current and voltage. These variables are directly proportional to active and reactive power and including them alongside power values can create confusion for the model, making it unclear whether to rely on the actual power measurements or the derived ones. Removing such overlapping features helps reduce data complexity and allows the model to focus on the most relevant inputs. Since this is a forecasting model where all remaining features are considered equally important, there's no need to assign different weights or perform further selection based on feature importance. During data processing, categorical information, like day names are converted into numerical form so the model can interpret it effectively. The entire dataset is then normalized by scaling values between 0 and 1, ensuring consistency across features and enhancing the overall training efficiency and model performance.

Given the time-series nature of load data and the objective of forecasting both the next day and the next week, a multi-horizon forecasting approach is adopted. This method allows the model to predict a full sequence of future values, such as the next 24 hours or 7 days, in a single pass, rather than predicting each time step individually. To achieve this, the input and output sequences are carefully structured. The processed data is then fed into a GRU (Gated Recurrent Unit) model, designed with multiple hidden layers and a hidden size of 128. This architecture is particularly effective at capturing temporal patterns and dependencies, making it a reliable and efficient choice for forecasting over multiple time horizons.[36]

The model training process focuses on fine-tuning the GRU network using carefully chosen parameters. It uses a learning rate of 0.003, the Adam optimizer for efficient weight updates, and the SmoothL1Loss function to measure prediction errors. The training is carried out over 7 epochs with a batch size of 16. Throughout the training, the loss is continuously monitored, and a loss curve is plotted to track how well the model is learning and to ensure proper convergence. Once training is complete, the model is saved and used to generate future load forecasts. The final output includes the model's predictions, visualizations of the results, and the relevant data stored for future reference.

In the forecasting loop, the same structured approach is followed, with the key difference being that the already trained GRU model is now used to generate predictions instead of undergoing training. The incoming historical data is first validated to meet the required format, then pre-processed in the same way—this includes feature selection, categorical encoding, and normalization. Once the data is prepared, it is passed through the trained GRU model, which outputs load forecasts for the specified horizons (next day or next week). The results are then post-processed, visualized, and saved, completing the end-to-end forecasting cycle efficiently using the pre-trained model. This structured approach highlights the importance of proper data handling and thoughtful model design in achieving accurate and dependable load forecasting.[37]

5.4 HMI: Load Forecasting

The forecasting interface is designed to simplify the user's interaction with the GRU-based prediction model. Built using Streamlit, it provides an intuitive two-tab layout that enables

both real-time inference and model training functionalities.

In the Generate Forecasts (figure25) tab, users can upload historical energy consumption data in CSV format and select the desired prediction horizon—either the next day (24 hours) or the next week (7 days). Once uploaded, the system automatically validates the frequency, resamples the data to ensure one-minute intervals, and normalises the input features. After this preprocessing, the pre-trained GRU model generates future predictions for active and reactive power values across all three phases. These results are visualised through time-series plots and can be downloaded as a CSV file for further analysis.

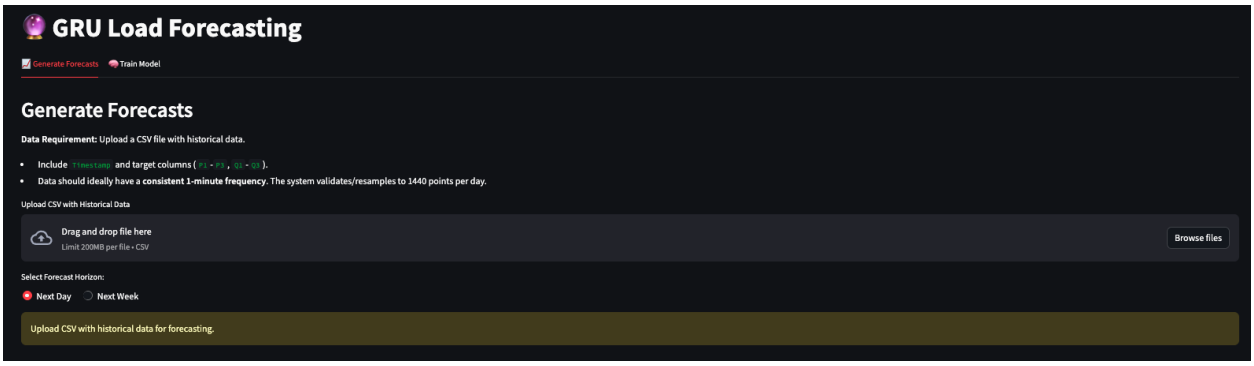


Figure 25: Generate Forecasts Tab. Users can upload historical load data and generate energy forecasts for the next day or week using a pre-trained GRU model. Forecasts are visualised as time-series plots and can be downloaded for further use.

The Train Model (figure 26) tab allows users to retrain the GRU model on custom data. After uploading the dataset, users can configure training parameters such as the number of epochs, learning rate, and batch size. Training progress is visualised in real time using loss curves, providing clear feedback on model performance. Upon successful training, the model weights are saved and made available for immediate forecasting use.

Together, these two tabs provide a comprehensive and user-friendly platform for industrial load forecasting using deep learning.

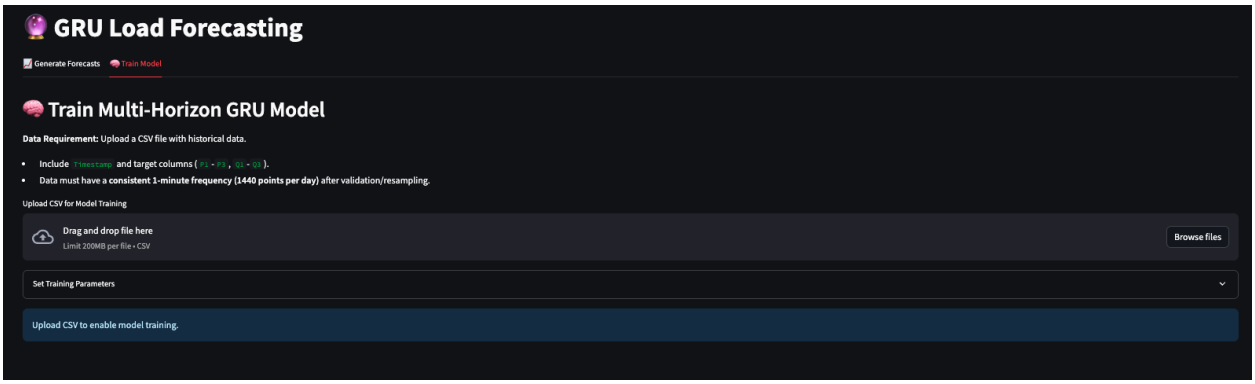


Figure 26: Train Model Tab. This interface allows users to retrain the GRU forecasting model on new data. Training progress is visualised with real-time loss plots, and the updated model can be saved for future forecasting tasks.

6 Implementation in the Laboratory Environment

To enable replicability and future deployment in industrial or academic setups, the Cyber Physical Energy Management System (CPEMS) is implemented using a modular architecture that supports both simulation and forecasting. This section outlines the folder structure, required software, and step-by-step setup for running the system in a laboratory environment.

6.1 Software and Dependencies

The following software and packages are required to run the system:

- **Python** (version 3.9 or higher) with packages:
  - `streamlit`, `pandas`, `numpy`
  - `torch`, `matplotlib`
  - `influxdb-client`, `scikit-learn` (optional)
- **MATLAB** (with Simulink) for simulation
- **MATLAB Engine API for Python** to enable MATLAB integration
- **InfluxDB Cloud (Free Tier)** for storing and querying time-series data

To enable simulation, the MATLAB Engine for Python must be installed. This can be done by executing the following from the MATLAB installation directory:

```
cd "matlabroot/extern/engines/python"
python setup.py install
```

Replace `matlabroot` with your local MATLAB installation path.

6.2 Folder Structure

The project is organized with the following structure:

```
CPEMS/
 pages/
    1_Forecasting.py          # GRU Forecasting interface
    2_Simulation.py           # MATLAB simulation + InfluxDB visualisation
 Run_Simulink_Function.m       # MATLAB function to execute simulation
 Switch.slx                    # Simulink model file
 main.py                       # HMI home page in Streamlit
```

6.3 Steps to Run the System

1. **Ensure MATLAB and Simulink are installed** and that the MATLAB Engine API is functional. You can test it by running:

```
import matlab.engine
eng = matlab.engine.start_matlab()
```

2. **Configure the system:** Update the InfluxDB URL, token, and organisation parameters in the code. Also, set the MATLAB script path to point to the correct location of `Run_Simulink_Function.m` and `Switch.slx`.

3. **Launch the application:**

   ```
   streamlit run main.py
   ```

   This will start the Streamlit interface in a web browser, where users can access:

   - **Load Forecasting** – to train the GRU model and generate forecasts
   - **Industrial Profile** – to simulate energy loads and visualise results

6.4 Notes on Laboratory Setup

- The **simulation functionality** requires MATLAB to be locally installed and accessible via the Python API. The MATLAB files must be located in the specified path for successful integration.
- **Data is stored in Influxdb Cloud** (Free Tier), which provides reliable and scalable time-series data handling. An active internet connection is required for data upload and query.

7 System Validation and Future Work

7.1 Validation Criteria and Results

In order to validate the developed system, a consideration of the use selected use cases, load displaying and load forecasting, is conducted. Also scalability is taken into account.

Regarding the load displaying, the dashboard offers a good solution to view real-time data with details, such as harmonics, as well as aggregated data. It reacts fast to user inputs and even offers the function to restart the simulation with a different selection of machines. The rerunning of the simulation takes not too long, although it is not the speed of common Power BI tools.

Regarding the load forecasting, the algorithm manages to perform a good forecast based on historic data. However, the accuracy could be even higher with more training epochs used: For instance, it was observed that the model predicts slightly different load patterns over the same kind of week days. Also, sometimes the load on a weekend day was not predicted as purely base load. Increasing the number of training epochs will increase the time needed for training as well. Hence, the implementation of the training on a powerful server, see below, is recommended if the training is executed frequently. Concerning the use case, a training would probably be conducted not every day, but rather every week or month. On a server, the training could run overnight as well. The remaining components of the application however can run on a more or less powerful stationary PC as well.

The scalability is given in many points: Machines can be added to the Simulink simulation. The Python code is highly modular. The simulation time is assumed to slightly increase with every machine added. However, the machine selection by a parameter mapping becomes more complex with a higher number of modeled consumers: The parameter count needed to reflect all possible combinations will grow with two to the power of n. Hence, for rather large factories with a high number of machines, switching to alternative approaches can be considered. Such may be similar to the alternative daily cycle methods described as polynomial fits: Average runtime patterns for different machines can be implemented rather than a selection of individual machines.

7.2 Cloud based Training of the Forecasting Network

As mentioned before, the computational capabilities of the local machines used for development were not sufficient to train the neural network for the load forecasting. As a consequence, it is derived that also the local machine given in the laboratory, which has similar memory capabilities, will not be able to deliver proper results for the forecasting with a sufficient runtime. Therefore, a cloud based computing is explored as a viable solution. The following considerations have been made to specify the requirements for the server setup:

- The operating system can be Windows based or Linux. Python and Matlab are available for both. However, since the whole development has been performed in Windows, this is recommended.

- The memory must be sufficient to run the forecasting algorithm and should provide a significant benefit towards a local machine setup in terms of computation speed
- The server must be accessible from remote and be able to easily exchange the data with the main Python function running on the local lab computer.

7.3 Future Work

Besides the full implementation on a lab computer and the implementation of the training of the forecasting model on a remote server or virtual machine, further approaches of future work are listed below.

In the first chapter, further use cases have been worked out, such as the load optimization and the integration of renewable energy generation. Since these are growing fields of interest, taking the generation side into account can surely be subject to future work.

Also, right now non-linearity is not implemented in the Simulink model. Harmonics are inserted via the basic controlled voltage source. Generally, only synthesized data are used. Future work can include the integration of non-linearity in the simulation or the use of real-world data. Also, further metrics like Total Harmonic Distortion can be added to the data analysis. In addition, further machine types can be taken into account, including the individual observation of each machine by putting more measurement units. In order to observe characteristics like transient stability and oscillation of a machine, the sample rate could be increased even further to more than one data point per millisecond.

In addition, the two use case can be further integrated into one central code by reading the simulation data for the model training and load forecasting directly out of Influx DB rather than passing via a .csv file.

8 Conclusion

In summary, a cyber physical energy management system has been built up, sufficiently supporting the two selected use cases of Energy Monitoring and Energy Demand Forecasting. Instead of a initially planned grey-box model, it turned into a whit-box model, only using synthesized data generated by a Simulink simulation. The simulation results are processed and written into a database by the interaction of the Python main function, the Matlab code and the Simulink file. A GRU model is trained based on the simulation data, which are now passed via a .csv file. The model is able to predict the future loads accurately, with the constraint of a sufficient number of training epochs and time. In a Streamlit dashboard, the user can start the simulation and the forecasting based on selected parameters and the results of both use cases are visualized. The outputs of typical industry measurement devices (Siemens Sentron PAC 4200 and Beckhoff Twincat) are taken as a reference for the data format to prepare the the implementation of real-world data in the future.

Additional future work includes the implementation of further use cases, machines and metrics as well as the implementation in a lab environment with a powerful training computation source. Also, the two use cases can be code-wise combined into one by allowing direct data transfer between them.

References

[1] M. Prof. Dr. Freiburg and F. Prof. Dr. Hackelöer, "Case study module: Modelling and simulation of technical systems (msts) - 3. cyber-physical energy management system," *Case Study Assignments 2024 - Master of Automation and IT - TH Cologne - Internal Document*, 2024.

[2] G. J. Tsekouras, N. D. Hatziargyriou, and E. N. Dialynas, "Two-stage pattern recognition of load curves for classification of electricity customers," *IEEE Transactions on Power Systems*, vol. 22, no. 3, pp. 1120–1128, 2007.

[3] P. Waide and W. S. E. Ltd., "Introduction to the eu's energy efficiency directive," in *EU Energy and Climate Policy - Host: Leonardo Energy*, 2015.

[4] E. Union, "Directive 2012/27/eu of the european parliament and of the council of 25 october 2012 on energy efficiency, amending directives 2009/125/ec and 2010/30/eu and repealing directives 2004/8/ec and 2006/32/ec," *Official Journal of the European Union L315/1 EN*, 2012.

[5] A. Kluczek and P. Olszewski, "Energy audits in industrial processes," *ScienceDirect - Journal of Cleaner Production*, 2016.

[6] S. Noy, "Investor's willingness to pay for esg funds," *National Bureau of Economic Research - The Digest*, 2023.

[7] S. R. Department, "Global esg etf assets from 2006 to november 2023," tech. rep., Statista, 2024.

[8] F. T. Wenzel, "Mittelstand ist wenig digital - tausende unternehmen hinken bei der anwendung moderner it hinterher," *Rhein-Sieg Anzeiger - Am Wochenende - Nr. 301 SRS - 28.12.2024*, 2024.

[9] E. Commission, "Directive 2003/87/ec of the european parliament and of the council of 13 october 2003 establishing a scheme for greenhouse gas emission allowance trading within the community and amending council directive 96/61/ec," *Official Journal of the European Union L275/32 EN*, 2003.

[10] M. des Innern des Landes Nordrhein-Westfalen, "Verordnung zur umsetzung der solaranlagen-pflicht nach § 42a und § 48 absatz 1a der bauordnung für das land nordrhein-westfalen (solaranlagen-verordnung nordrhein-westfalen – san-vo nrw)," *Recht.NRW.DE - Geltende Gesetze und Verordnungen (SGV.NRW)*, 2024.

[11] NAU.CH, "load shedding: Stunden kein strom: Südafrikas weinindustrie leidet," *NAU.CH, Member of DPA*, 2023.

[12] Siemens AG, *Operating Instructions - SENTRON PAC4200 Power Monitoring Device*, 2012. Version 03, English (US).

[13] Beckhoff Automation GmbH Co. KG, "C6030 — ultra-compact industrial pc," 2024. Accessed: 2025-05-05.

[14] Eaton, "What is a non-linear load?," n.d. Accessed: 2025-05-05.

[15] MathWorks, "Controlled voltage source - simscape electrical," n.d. Accessed: 2025-05-05.

[16] Eaton, "Harmonics: Understanding and managing harmonic distortion in electrical power systems," n.d. Accessed: 2025-05-05.

[17] MathWorks, "Voltage measurement - simscape electrical," n.d. Accessed: 2025-05-05.

[18] MathWorks, "Current measurement - simscape electrical," n.d. Accessed: 2025-05-05.

[19] MathWorks, "Power measurement - simscape electrical," n.d. Accessed: 2025-05-05.

[20] V. Mehta and R. Mehta, *Principles of Power System*. S. Chand Publishing, revised edition ed., 2014.

[21] MathWorks, "To workspace - simulink," n.d. Accessed: 2025-05-05.

[22] MathWorks, "Matlab function - simulink," n.d. Accessed: 2025-05-05.

[23] Energy Efficiency Center, Oregon State University, "Common industrial motor types," 2024. Accessed: 2025-05-05.

[24] Rongke Group, "Industrial induction furnace." `https://www.rongkegroup.com/induction-furnace-industrial/`, n.d. Accessed: 2025-05-05.

[25] MathWorks, "Step." `https://de.mathworks.com/help/simulink/slref/step.html`, n.d. Accessed: 2025-05-05.

[26] V. Labs, "Data manipulation plugin for grafana — manual data entering and user input into dashboard," 2022. accessed 24.01.2025.

[27] Streamlit Inc., *Streamlit: The fastest way to build and share data apps*, 2024. Accessed: 2025-04-30.

[28] MathWorks, *MATLAB Engine API for Python*, 2024. Accessed: 2025-04-30.

[29] InfluxData, *InfluxDB Documentation*, 2024. Accessed: 2025-04-30.

[30] C. R. Atla, "Load forecasting: Methods & techniques," tech. rep., SAARC Energy Centre, 2018. Accessed: 2025-05-01.

[31] H. Mubarak, M. J. Sanjari, A. Abdellatif, and S. S., "Improved active and reactive energy forecasting using a stacking ensemble approach: Steel industry case study," *Energies*, vol. 16, no. 21, p. 7252, 2023. Accessed: 2025-05-01.

[32] N. Fidalgo and J. A. Lopes, "Forecasting active and reactive power at substations' transformers," in *IEEE Power Tech Conference Proceedings*, vol. 1, p. 6 pp., 2003.

[33] L. Babich, D. Svalov, A. Smirnov, and M. Babich, "Industrial power consumption forecasting methods comparison," *2019 Ural Symposium of Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT)*, 2019.

[34] J. Arce and E. Macabebe, "Real-time power consumption monitoring and forecasting using regression techniques and machine learning algorithms," *2019 IEEE International Conference on Internet of Things and Intelligence Systems (IoTaIS)*, 2019.

[35] M. Abumohsen, A. Y. Owda, and M. Owda, "Electrical load forecasting using lstm, gru, and rnn algorithms," *Energies*, vol. 16, no. 5, p. 2283, 2023. Accessed: 2025-05-01.

[36] PyTorch Contributors, *torch.nn.GRU — PyTorch 2.7 Documentation*. PyTorch, 2025. Accessed: 2025-05-01.

[37] GeeksforGeeks Contributors, "Gated recurrent unit networks." `https://www.geeksforgeeks.org/gated-recurrent-unit-networks/`, April 2025. Accessed: 2025-05-01.

A Appendix

A.1 Full Python Code

A.1.1 Python Code for Main Home Tab

```
import streamlit as st

st.set_page_config(
    page_title="CPEMS Home",
    layout="wide", # Use wide layout
    initial_sidebar_state="expanded" # Keep sidebar open initially
)

# --- Page Configuration ---
# Paths based on the filenames 'pages/1_Forecasting.py' and 'pages/2_Simulation.py'
FORECASTING_PAGE_PATH = "/Forecasting"
SIMULATION_PAGE_PATH = "/Simulation"

# --- Main Page Content ---
st.title("    Cyber Physical Energy Management System    ")
st.markdown("---") # Separator line

st.subheader("Welcome!")
st.markdown("""
Navigate through the different modules of the CPEMS using the sidebar on the left, or click one of the boxes below.
- **Forecasting:** Train the GRU model and generate multi-horizon energy forecasts.
- **Industrial Profile:** Run MATLAB/Simulink simulations and visualize data from InfluxDB.
""")

st.markdown("---")

# --- Clickable Visual Boxes (using columns and markdown with HTML links) ---
col1, col2 = st.columns(2)

# Common style for the links to make them behave like block elements and remove underlines
link_style = "text-decoration: none; color: inherit; display: block; height: 100%;"

# Common style for the inner div content box
# Added flexbox properties for better vertical centering of content
div_common_style = "border-radius: 10px; padding: 20px; text-align: center; height: 200px; display: flex; flex-direction: column; justify-content: center; align-items: center;"

with col1:
    # Specific styles for the forecasting box
    forecasting_div_style = f"{div_common_style} border: 2px solid #4CAF50
```

```
    ; background-color: #f0fff0;"
    st.markdown(
        f"""
        <a href="{FORECASTING_PAGE_PATH}" target="_self" style="{link_style}">
            <div style="{forecasting_div_style}">
                <h2 style="color: #4CAF50; margin-bottom: 15px;">     Load Forecasting</h2>
                <p style="color: #333; margin-bottom: 10px;">Access the forecasting module to predict future energy consumption using the GRU model.</p>
                <p style="color: #555; font-style: italic; font-size: 0.9em;">Click this box to navigate</p>
            </div>
        </a>
        """,
        unsafe_allow_html=True
    )

with col2:
    # Specific styles for the simulation box
    simulation_div_style = f"{div_common_style} border: 2px solid #2196F3; background-color: #f0f8ff;"
    st.markdown(
        f"""
        <a href="{SIMULATION_PAGE_PATH}" target="_self" style="{link_style}">
            <div style="{simulation_div_style}">
                <h2 style="color: #2196F3; margin-bottom: 15px;">Industrial Profile</h2>
                <p style="color: #333; margin-bottom: 10px;">Run MATLAB simulations, upload data to InfluxDB, and visualize simulation results.</p>
                <p style="color: #555; font-style: italic; font-size: 0.9em;">Click this box to navigate</p>
            </div>
        </a>
        """,
        unsafe_allow_html=True
    )

st.markdown("---") # Separator line at the bottom

# Update sidebar message
st.sidebar.success("Select a page above or click a main panel box.")

# Add any other information you want on the main landing page below
# st.info("System Status: OK")
```

Listing 1: Main Home Page Code

A.1.2 Python Code for Forecasting Tab

```python
# --- Imports for Forecasting ---
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import streamlit as st
import matplotlib.pyplot as plt
from datetime import datetime, timedelta, time # Import time
# Note: LabelEncoder is not used, relying on pd.Categorical or numeric days
from torch.utils.data import Dataset, DataLoader
import os
import math
import sys # To check python version if needed

# --- Configuration & Setup ---
st.set_page_config(page_title="GRU Forecasting", layout="wide")
st.title("     GRU Load Forecasting")

# Device setup (GPU if available)
@st.cache_resource # Cache the device resource
def get_device():
    dev = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    # print(f"Forecasting using device: {dev}") # Add print statement for debugging/confirmation
    return dev
device = get_device()
st.sidebar.info(f"Using device: {device}") # Display device info in sidebar


# Paths for model saving and loading
MODEL_PATH = "multi_horizon_gru_model.pth" # Path for the GRU model weights

# Define required input columns for the model and target columns for prediction
# Timestamp is implicitly required by the new load function for resampling
REQUIRED_INPUT_COLUMNS = ["P1", "P2", "P3", "Q1", "Q2", "Q3", "Day"]
TARGET_COLUMNS = ["P1", "P2", "P3", "Q1", "Q2", "Q3"] # Columns the model predicts

# GRU Model Parameters
INPUT_SIZE = len(REQUIRED_INPUT_COLUMNS) # Number of features in input sequence
TARGET_SIZE = len(TARGET_COLUMNS)      # Number of features to predict
HIDDEN_SIZE = 128                      # Number of features in GRU hidden state
NUM_GRU_LAYERS = 2                     # Number of GRU layers
SEQUENCE_LENGTH = 1440                 # Input sequence length (e.g., 1 day = 1440 minutes)
```

```
DAY_HORIZON = 1440                      # Prediction horizon for 1 day
WEEK_HORIZON = 1440 * 7                 # Prediction horizon for 7 days


# --- GRU Model Definition ---
class MultiHorizonGRU(nn.Module):
    """
    GRU model predicting multiple horizons (1-day and 7-day) directly.
    """
    def __init__(self, input_size, hidden_size, num_gru_layers, day_horizon, week_horizon, target_size):
        super(MultiHorizonGRU, self).__init__()
        dropout_rate = 0.2 if num_gru_layers > 1 else 0
        self.gru = nn.GRU(input_size, hidden_size, num_layers=num_gru_layers,
                          batch_first=True, dropout=dropout_rate)
        # Linear layers to map GRU output to the flattened multi-horizon predictions
        self.fc_day = nn.Linear(hidden_size, day_horizon * target_size)
        self.fc_week = nn.Linear(hidden_size, week_horizon * target_size)
        # Store dimensions for reshaping
        self.day_horizon = day_horizon
        self.week_horizon = week_horizon
        self.target_size = target_size

    def forward(self, x):
        # x shape: (batch, seq_len, input_size)
        gru_out, h_n = self.gru(x)
        # Use the last hidden state of the last layer for prediction
        x_last = h_n[-1] # shape: (batch, hidden_size)
        # Predict day horizon
        out_day_flat = self.fc_day(x_last) # shape: (batch, day_horizon * target_size)
        out_day = out_day_flat.view(-1, self.day_horizon, self.target_size) # shape: (batch, day_horizon, target_size)
        # Predict week horizon
        out_week_flat = self.fc_week(x_last) # shape: (batch, week_horizon * target_size)
        out_week = out_week_flat.view(-1, self.week_horizon, self.target_size) # shape: (batch, week_horizon, target_size)
        return out_day, out_week

# --- Load or Initialize Model ---
@st.cache_resource # Cache the loaded model object
def load_model(model_path, device):
    """Loads the GRU model state dict, caching the model instance."""
    model_instance = MultiHorizonGRU(INPUT_SIZE, HIDDEN_SIZE, NUM_GRU_LAYERS, DAY_HORIZON, WEEK_HORIZON, TARGET_SIZE).to(device)
    if os.path.exists(model_path):
        try:
            model_instance.load_state_dict(torch.load(model_path, map_location=device))
            st.sidebar.success("Loaded pre-trained GRU model.")
        except Exception as e:
```

```
            st.sidebar.error(f"Error loading GRU state: {e}")
            st.sidebar.warning("Using untrained model.")
    else:
        st.sidebar.warning("No pre-trained GRU model found at specified path.")
    return model_instance

# Load the model once using the cached function
model = load_model(MODEL_PATH, device)

# Define the loss function (used during training)
criterion = nn.SmoothL1Loss()


# --- Data Handling Functions ---

def create_multi_horizon_sequences(data, target_data, seq_length, day_horizon, week_horizon):
    """ Creates input sequences (X) and corresponding multi-horizon target sequences. """
    X, y_day, y_week = [], [], []
    max_horizon = max(day_horizon, week_horizon)
    # Ensure enough data for at least one sequence + longest horizon
    if len(data) < seq_length + max_horizon:
        st.error(f"Insufficient data length ({len(data)}) after processing to create sequences. "
                 f"Need at least {seq_length + max_horizon} rows.")
        return np.array([]), np.array([]), np.array([])

    # Iterate through the data to create sequences
    num_sequences = len(data) - seq_length - max_horizon + 1
    if num_sequences <= 0:
         st.error(f"Cannot create sequences with current data length ({len(data)}), sequence length ({seq_length}), "
                  f"and max horizon ({max_horizon}).")
         return np.array([]), np.array([]), np.array([])

    for i in range(num_sequences):
        X.append(data[i : i + seq_length])
        y_day.append(target_data[i + seq_length : i + seq_length + day_horizon])
        y_week.append(target_data[i + seq_length : i + seq_length + week_horizon])

    return np.array(X), np.array(y_day), np.array(y_week)

# --- Data Preprocessing (MODIFIED: Returns last timestamp) ---
def load_and_prepare_data(file_path, required_columns, target_columns):
    """
    Loads data, gets last timestamp, validates/resamples, preprocesses, normalizes.
    Returns: normalized input data, normalized target data, scalers, last_known_timestamp.
    """
```

```
    last_known_timestamp = None # Initialize
    try:
        if hasattr(file_path, 'getvalue'):
            # Reset buffer if it's an UploadedFile object that might have been read
            try:
                file_path.seek(0)
            except: pass # Ignore if seek is not available or fails
            df = pd.read_csv(file_path)
        else:
            df = pd.read_csv(str(file_path))
        # st.info(f"Loaded CSV: {df.shape[0]} rows, {df.shape[1]} columns.") # Less verbose
    except FileNotFoundError:
        st.error(f"Error: File not found at {file_path}")
        return None, None, None, None
    except Exception as e:
        st.error(f"Error reading CSV file: {e}")
        return None, None, None, None

    # --- 1. Process Timestamp and Find Last Valid One ---
    if 'Timestamp' not in df.columns:
        st.error("Validation Failed: Missing required 'Timestamp' column.")
        return None, None, None, None
    try:
        df['Timestamp'] = pd.to_datetime(df['Timestamp'], errors='coerce')
        original_rows = len(df)
        df.dropna(subset=['Timestamp'], inplace=True)
        rows_dropped = original_rows - len(df)
        if rows_dropped > 0:
            st.warning(f"Removed {rows_dropped} rows with invalid Timestamp formats.")
        if df.empty:
            st.error("No valid Timestamps found after conversion.")
            return None, None, None, None

        # *** GET LAST TIMESTAMP ***
        last_known_timestamp = df['Timestamp'].max()
        st.info(f"Last timestamp found in data: {last_known_timestamp}")

        df.set_index('Timestamp', inplace=True, drop=False) # Keep Timestamp column if needed
        df.sort_index(inplace=True)
    except Exception as e:
        st.error(f"Could not process 'Timestamp' column: {e}")
        return None, None, None, None # Return None for timestamp as well

    # --- 2. Frequency Validation & Resampling per Day ---
    processed_days_data = []
    unique_days = df.index.normalize().unique()
    # st.info(f"Data spans {len(unique_days)} days. Validating frequency (1440 points/day)...") # Less verbose
    validation_or_resampling_failed = False
```

```
    # Use st.expander for potentially long validation output, keep it collapsed by default
    with st.expander("Frequency Validation Details", expanded=False):
        with st.spinner("Validating data frequency per day..."):
            for day_date in unique_days:
                day_data = df[df.index.date == day_date.date()]
                day_point_count = day_data.shape[0]
                if day_point_count == 0: continue
                if day_point_count < 1440:
                    st.error(f"Validation Failed: Day {day_date.date()} has {day_point_count} points (requires 1440).")
                    validation_or_resampling_failed = True; break
                elif day_point_count > 1440:
                    st.write(f"Resampling day {day_date.date()} ({day_point_count} points)...")
                    try:
                        numeric_cols = day_data.select_dtypes(include=np.number).columns
                        day_resampled = day_data[numeric_cols].resample('T').mean()
                        day_resampled = day_resampled.ffill().bfill()
                        if len(day_resampled) != 1440:
                            st.error(f"Resampling Error: Day {day_date.date()} resulted in {len(day_resampled)} points.")
                            validation_or_resampling_failed = True; break
                        processed_days_data.append(day_resampled)
                    except Exception as resample_e:
                        st.error(f"Resampling Error for day {day_date.date()}: {resample_e}")
                        validation_or_resampling_failed = True; break
                else:
                     # st.write(f"Day {day_date.date()} has exactly 1440 points (OK).") # Less verbose
                     processed_days_data.append(day_data) # Append original day_data if count is correct
            if validation_or_resampling_failed:
                 st.error("Frequency validation or resampling failed. Cannot proceed.")
                 return None, None, None, last_known_timestamp
            # else: st.success("Frequency checks passed.") # Less verbose

    if not processed_days_data:
         st.error("No valid data days found after processing.")
         return None, None, None, last_known_timestamp
    df_processed = pd.concat(processed_days_data)
    df_processed.sort_index(inplace=True)
    # st.success(f"Frequency processing complete. Final dataset: {df_processed.shape[0]} rows.") # Less verbose

    # --- 3. Derive 'Day' column (0-6) AFTER resampling ---
    # Use index which is the Timestamp after resampling
    df_processed['Day'] = df_processed.index.dayofweek

    # --- 4. Check Required Columns exist AFTER potential resampling ---
```

```
    # Ensure P1-P3, Q1-Q3, Day are present
    final_check_cols = required_columns[:] # Copy list
    missing_cols = [col for col in final_check_cols if col not in df_processed.columns]
    if missing_cols:
        # Check if original df had them before resampling potentially dropped non-numeric cols
        orig_missing = [col for col in final_check_cols if col not in df.columns and col != 'Day']
        if orig_missing:
            st.error(f"Input Error: Original CSV missing required numeric columns: {', '.join(orig_missing)}")
        else:
            st.error(f"Processing Error: Missing required columns after resampling/processing: {', '.join(missing_cols)}. Ensure they were numeric.")
        return None, None, None, last_known_timestamp

    # --- 5. Select Final Columns & Final NaN Check ---
    # Select only the columns needed for the model input features
    df_final_input = df_processed[final_check_cols].copy()
    if df_final_input.isnull().values.any():
        st.warning("NaNs detected late in processing. Filling with ffill then zero.")
        df_final_input = df_final_input.ffill().fillna(0) # Fill remaining NaNs
    if df_final_input.isnull().values.any():
        st.error("Data still contains NaNs after attempting to fill. Cannot proceed.")
        return None, None, None, last_known_timestamp

    # --- 6. Normalization ---
    try:
        # Calculate scalers based on the final input data
        # Only include numeric columns for min/max calculation
        numeric_cols_for_scaling = df_final_input.select_dtypes(include=np.number).columns
        data_min = df_final_input[numeric_cols_for_scaling].min()
        data_max = df_final_input[numeric_cols_for_scaling].max()
        data_range = (data_max - data_min).replace(0, 1) # Avoid division by zero

        # Apply normalization only to numeric columns found
        df_normalized = df_final_input.copy() # Start with a copy
        for col in data_range.index: # Iterate through columns with valid range
            df_normalized[col] = (df_final_input[col] - data_min[col]) / data_range[col]


        # Prepare outputs
        # Scalers should include all required input columns if they were numeric
        scalers = {col: {'min': df_final_input[col].min(), 'max':
```

```
    df_final_input[col].max()}
                    for col in final_check_cols if col in df_final_input.select_dtypes(include=np.number).columns}


        # Ensure df_normalized has all required columns before creating numpy array
        if not all(col in df_normalized.columns for col in required_columns):
            missing_norm_cols = [col for col in required_columns if col not in df_normalized.columns]
            st.error(f"Normalization Error: Missing columns after normalization: {', '.join(missing_norm_cols)}")
            return None, None, None, last_known_timestamp

        # Convert to numpy array using the final required column order
        data_normalized_all = df_normalized[required_columns].values.astype(np.float32)

        # Ensure target columns exist in the normalized df before selecting
        if not all(col in df_normalized.columns for col in target_columns):
            st.error(f"Target columns ({', '.join(target_columns)}) not found after normalization.")
            return None, None, None, last_known_timestamp
        data_normalized_target = df_normalized[target_columns].values.astype(np.float32)

        # st.success("Data loading, validation, processing, and normalization complete.") # Less verbose
        # *** MODIFIED RETURN ***
        return data_normalized_all, data_normalized_target, scalers, last_known_timestamp
    except Exception as norm_e:
        st.error(f"Error during data normalization: {norm_e}")
        return None, None, None, last_known_timestamp
# ------------------------------------------------------------------------------


class MultiTargetDataset(Dataset):
    """ Custom PyTorch Dataset for multi-horizon forecasting targets. """
    def __init__(self, X, y_day, y_week):
        self.X = X
        self.y_day = y_day
        self.y_week = y_week
    def __len__(self):
        return len(self.X)
    def __getitem__(self, idx):
        # Return data as tensors for DataLoader
        return torch.tensor(self.X[idx], dtype=torch.float32), \
               torch.tensor(self.y_day[idx], dtype=torch.float32), \
```

```
                torch.tensor(self.y_week[idx], dtype=torch.float32)

# --- Forecasting Function ---
def forecast_multi_horizon(model, initial_data_sequence, device):
    """ Generates fixed 1-day and 7-day forecasts using the trained GRU model. """
    model.eval() # Set model to evaluation mode
    with torch.no_grad(): # Disable gradient calculations for inference
        input_tensor = torch.tensor(initial_data_sequence, dtype=torch.float32).unsqueeze(0).to(device)
        forecast_day_tensor, forecast_week_tensor = model(input_tensor)
        forecast_day_np = forecast_day_tensor.squeeze(0).cpu().numpy()
        forecast_week_np = forecast_week_tensor.squeeze(0).cpu().numpy()
    return forecast_day_np, forecast_week_np


# --- Streamlit UI Sections ---

def forecasting_ui():
    """ Defines the Streamlit UI for the Forecasting section. """
    st.header("Generate Forecasts")
    st.markdown("""
    **Data Requirement:** Upload a CSV file with historical data.
    - Include `Timestamp` and target columns (`P1`-`P3`, `Q1`-`Q3`).
    - Data should ideally have a **consistent 1-minute frequency**. The system validates/resamples to 1440 points per day.
    """)

    uploaded_forecast_file = st.file_uploader("Upload CSV with Historical Data", type=["csv"], key="forecast_uploader")
    forecast_option = st.radio("Select Forecast Horizon:", ["Next Day", "Next Week"], key="forecast_horizon_choice", horizontal=True)

    if uploaded_forecast_file:
        st.info(f"Model requires last {SEQUENCE_LENGTH} data points as input.")

        # *** MODIFIED CALL: Receive last_known_timestamp ***
        data_norm_all, _, scalers, last_known_timestamp = load_and_prepare_data(
            uploaded_forecast_file,
            REQUIRED_INPUT_COLUMNS,
            TARGET_COLUMNS
        )

        if data_norm_all is None or scalers is None:
            st.error("Failed to load/process forecast data. Check messages above.")
            return # Use return instead of st.stop() within function

        if len(data_norm_all) < SEQUENCE_LENGTH:
            st.error(f"Insufficient data after processing. Need >= {SEQUENCE_LENGTH} rows, got {len(data_norm_all)}.")
            return
```

```
        initial_sequence = data_norm_all[-SEQUENCE_LENGTH:]
        data_available = True
        # st.success("Data ready for forecasting.") # Less verbose

        # --- Check if a valid last timestamp was found ---
        if pd.isna(last_known_timestamp):
            st.warning("Could not determine last timestamp from file. Forecast time reference will default to current time.")
            reference_time = datetime.now() # Fallback reference
            last_day_numeric = reference_time.weekday()
            using_fallback_time = True
        else:
            reference_time = last_known_timestamp # Use actual last timestamp
            last_day_numeric = reference_time.weekday()
            using_fallback_time = False

    else:
        st.warning("Upload CSV with historical data for forecasting.")
        data_available = False
        # return # Allow UI elements below button to render

    # Proceed only if data is available
    if data_available:
        if forecast_option == "Next Day":
            horizon_steps = DAY_HORIZON
            horizon_label = "Next Day"
            horizon_file_part = "next_24_hours"
        else: # Next Week
            horizon_steps = WEEK_HORIZON
            horizon_label = "Next Week"
            horizon_file_part = "next_7_days"

        if st.button(f"     Generate Forecast for {horizon_label}", key="generate_forecast_button"):
            with st.spinner(f"Generating {horizon_label} forecast..."):
                forecast_day_norm, forecast_week_norm = forecast_multi_horizon(
                    model,
                    initial_sequence,
                    device
                )
                if forecast_option == "Next Day":
                    selected_forecast_norm = forecast_day_norm
                else:
                    selected_forecast_norm = forecast_week_norm

            st.success(f"    Forecast for {horizon_label} generated!")

            # --- Denormalization ---
            try:
                forecast_df_denormalized = pd.DataFrame(selected_forecast_norm, columns=TARGET_COLUMNS)
```

```
                for col in TARGET_COLUMNS:
                    if col in scalers:
                        min_val = scalers[col]['min']
                        max_val = scalers[col]['max']
                        range_val = max_val - min_val
                        if range_val != 0:
                            forecast_df_denormalized[col] = forecast_df_denormalized[col] * range_val + min_val
                        else:
                            forecast_df_denormalized[col] = min_val
                    else: st.warning(f"Scaler info missing for '{col}'.")
                for col in TARGET_COLUMNS:
                    forecast_df_denormalized[col] = forecast_df_denormalized[col].round(4)
            except Exception as denorm_e:
                st.error(f"Error during forecast denormalization: {denorm_e}")
                return # Stop processing this forecast if denorm fails

            # *** MODIFIED: Day Labels & Timestamps BASED ON LAST DATA POINT (or fallback) ***
            days_of_week = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
            # 'reference_time' and 'last_day_numeric' are set based on availability of last_known_timestamp

            forecast_title = ""
            xlabel_plot = f"Time Steps (from start of forecast)" # Default x-axis label
            plot_start_time = None

            try:
                if forecast_option == "Next Day":
                    # Start plot/timestamps at 00:00 of the day AFTER the reference time's day
                    next_day_date = reference_time.date() + timedelta(days=1)
                    plot_start_time = datetime.combine(next_day_date, time(0, 0)) # Start at midnight
                    next_day_index = (last_day_numeric + 1) % 7
                    forecast_day_label = days_of_week[next_day_index]
                    ref_time_str = reference_time.strftime('%Y-%m-%d %H:%M')
                    forecast_title = f"24h Forecast (For {forecast_day_label}, starting {plot_start_time.strftime('%Y-%m-%d %H:%M')})"
                    if using_fallback_time: forecast_title += f" - Ref: Current Time {ref_time_str}"
                    else: forecast_title += f" - Ref: Data End {ref_time_str}"

                else: # Next Week
                    # Start plot/timestamps 1 minute after the reference time
```

```
                    plot_start_time = reference_time + timedelta(minutes
=1)
                    start_day_index = (last_day_numeric + 1) % 7
                    forecast_days = [days_of_week[(start_day_index + i) %
7] for i in range(7)]
                    ref_time_str = reference_time.strftime('%Y-%m-%d %H:%M
')
                    forecast_title = f"7-Day Forecast ({forecast_days[0]}
to {forecast_days[-1]}, starting {plot_start_time.strftime('%Y-%m-%d %H
:%M')})"
                    if using_fallback_time: forecast_title += f" - Ref:
Current Time {ref_time_str}"
                    else: forecast_title += f" - Ref: Data End {
ref_time_str}"


                # Generate timestamp range for the forecast period with 1-
minute frequency
                forecast_timestamps = pd.date_range(start=plot_start_time,
 periods=horizon_steps, freq='T')

                # Add timestamp column to the DataFrame
                forecast_df_denormalized['Timestamp'] =
forecast_timestamps
                cols = ['Timestamp'] + TARGET_COLUMNS
                forecast_df_denormalized = forecast_df_denormalized[cols]
                xlabel_plot = "Timestamp" # Update plot label

            except Exception as e:
                 st.warning(f"Could not generate accurate timestamps/
labels: {e}. Plot will use time steps.")
                 # If title wasn't set due to error, set a default
                 if not forecast_title: forecast_title = f"{horizon_label}
 Forecast"


            # --- Display Results ---
            st.subheader("     Denormalized Forecast Sample")
            st.dataframe(forecast_df_denormalized.head())

            # --- Plotting ---
            st.subheader("     Forecast Visualization (Denormalized)")
            try:
                fig, ax = plt.subplots(figsize=(14, 7))
                x_axis = forecast_df_denormalized['Timestamp'] if '
Timestamp' in forecast_df_denormalized else forecast_df_denormalized.
index
                for col in TARGET_COLUMNS:
                    if col in forecast_df_denormalized:
                        ax.plot(x_axis, forecast_df_denormalized[col],
label=col)
                ax.legend()
                ax.set_xlabel(xlabel_plot)
                ax.set_ylabel("Predicted Values")
```

```
                ax.set_title(forecast_title) # Use updated title
                plt.xticks(rotation=45)
                plt.tight_layout()
                st.pyplot(fig)
            except Exception as plot_e:
                st.error(f"Error generating forecast plot: {plot_e}")


            # --- Download Button ---
            ref_time_str_file = reference_time.strftime('%Y%m%d_%H%M')
            csv_filename = f"forecast_{horizon_file_part}_ref_{ref_time_str_file}.csv"
            try:
                csv_data = forecast_df_denormalized.to_csv(index=False).encode('utf-8')
                st.download_button(
                    label=f"     Download Forecast ({horizon_label})",
                    data=csv_data,
                    file_name=csv_filename,
                    mime="text/csv",
                    key="download_forecast_button"
                )
            except Exception as e:
                 st.error(f"Failed to prepare download link: {e}")


def training_ui():
    """ Defines the Streamlit UI for the Model Training section. """
    st.header("     Train Multi-Horizon GRU Model")
    st.markdown("""
    **Data Requirement:** Upload a CSV file with historical data.
    - Include `Timestamp` and target columns (`P1`-`P3`, `Q1`-`Q3`).
    - Data must have a **consistent 1-minute frequency (1440 points per day)** after validation/resampling.
    """)

    uploaded_file = st.file_uploader("Upload CSV for Model Training", type=["csv"], key="train_uploader")

    with st.expander("Set Training Parameters"):
        num_epochs = st.number_input("Number of Epochs", min_value=1, max_value=1000, value=10, step=1, key="train_epochs")
        learning_rate = st.number_input("Learning Rate", min_value=1e-6, max_value=1e-1, value=0.001, step=1e-4, format="%.4f", key="train_lr")
        batch_size = st.selectbox("Batch Size", options=[8, 16, 32, 64], index=1, help="Number of sequences per training batch.", key="train_batch_size")

    if uploaded_file:
        if st.button("     Train GRU Model", key="train_model_button"):
            st.info("Loading and preparing training data...")
            # Load and preprocess - Ignore last timestamp return value for training
            data_norm_all, data_norm_target, _ , _ = load_and_prepare_data
```

```
    (
                uploaded_file,
                REQUIRED_INPUT_COLUMNS,
                TARGET_COLUMNS
            )

            if data_norm_all is None or data_norm_target is None:
                 st.error("Failed to load/process training data.")
                 return # Use return instead of st.stop()

            st.info("Creating training sequences...")
            X_train, y_day_train, y_week_train =
    create_multi_horizon_sequences(
                data_norm_all,
                data_norm_target,
                SEQUENCE_LENGTH,
                DAY_HORIZON,
                WEEK_HORIZON
            )

            if X_train.shape[0] == 0: return # Error handled in function

            st.success(f"Created {X_train.shape[0]} training sequences.")

            train_dataset = MultiTargetDataset(X_train, y_day_train,
    y_week_train)
            train_dataloader = DataLoader(train_dataset, batch_size=
    batch_size, shuffle=True, drop_last=True)

            model_to_train = model # Use globally loaded model
            model_to_train.to(device)
            optimizer = optim.Adam(model_to_train.parameters(), lr=
    learning_rate)

            st.info(f"Starting GRU training for {num_epochs} epochs...")
            model_to_train.train()
            progress_bar = st.progress(0.0)
            status_text = st.empty()
            loss_history = []
            chart_placeholder = st.empty()
            training_successful = True

            try:
                for epoch in range(num_epochs):
                    epoch_loss = 0.0
                    processed_batches = 0
                    num_batches = len(train_dataloader)
                    for i, batch_data in enumerate(train_dataloader):
                        try:
                            inputs, target_day, target_week = [d.to(device
    ) for d in batch_data]
                            out_day, out_week = model_to_train(inputs)
                            loss_day = criterion(out_day, target_day)
                            loss_week = criterion(out_week, target_week)
```

```
                            loss = loss_day + loss_week
                            if torch.isnan(loss) or torch.isinf(loss):
    continue
                            optimizer.zero_grad(); loss.backward();
    optimizer.step()
                            epoch_loss += loss.item(); processed_batches
    += 1
                        except Exception as batch_e:
                            st.warning(f"Batch {i+1} error: {batch_e}");
    continue

                    if processed_batches > 0:
                        avg_epoch_loss = epoch_loss / processed_batches
                        loss_history.append(avg_epoch_loss)
                        progress = (epoch + 1) / num_epochs
                        progress_bar.progress(progress)
                        status_text.text(f"Epoch {epoch + 1}/{num_epochs},
     Avg Loss: {avg_epoch_loss:.6f}")
                        loss_df = pd.DataFrame({'Epoch': range(1, epoch +
    2), 'Average Loss': loss_history})
                        chart_placeholder.line_chart(loss_df.set_index('
    Epoch'))
                    else: st.warning(f"Epoch {epoch+1} had no processed
    batches.")

            except Exception as train_e:
                 st.error(f"Training loop error: {train_e}");
    training_successful = False

            progress_bar.empty()
            if training_successful and loss_history:
                status_text.text(f"Training finished. Final Avg Loss: {
    loss_history[-1]:.6f}")
                st.success("    GRU Model training completed!")
                st.info(f"Saving trained model weights to {MODEL_PATH}..."
    )
                try:
                    model_to_train.to('cpu')
                    torch.save(model_to_train.state_dict(), MODEL_PATH)
                    st.success(f"    Model weights saved successfully!")
                    model_to_train.to(device)
                    st.cache_resource.clear(); st.info("Model cache
    cleared.")
                except Exception as e: st.error(f"Error saving model: {e}"
    )
            elif training_successful: status_text.text("Training loop
    finished, but no loss recorded.")
            else: status_text.text("Training stopped due to error.")
     else:
          st.info("Upload CSV to enable model training.")


# --- Main App Structure for this page ---
tab1, tab2 = st.tabs(["     Generate Forecasts", "     Train Model"])
```

```
with tab1: forecasting_ui()
with tab2: training_ui()
```

Listing 2: Streamlit Forecasting Tab Code

A.1.3 Python Code for Simulation Tab

```
# --- Imports ---
import streamlit as st
import pandas as pd
import numpy as np
import influxdb_client
from influxdb_client.client.write_api import SYNCHRONOUS
from datetime import datetime, timedelta, time # Import time for time_input
import os # Import os for file path operations
import sys # To check Python version

# --- MATLAB Availability Check ---
try:
    import matlab.engine
    MATLAB_AVAILABLE = True
except ImportError:
    MATLAB_AVAILABLE = False

# --- Original Logic Classes (Adapted for Integration) ---

class MatlabDataLoader:
    """ Handles MATLAB simulation execution and data writing. """
    def __init__(self, condition, matlab_script_path, influx_url, token, org, bucket, max_iterations=5000):
        self.condition = condition
        self.matlab_script_path = matlab_script_path
        self.bucket = bucket
        self.org = org
        self.token = token # Store token for writing
        self.max_iterations = max_iterations
        self.client = influxdb_client.InfluxDBClient(url=influx_url, token=self.token, org=org, verify_ssl=False)
        self.eng = None
        self.function_output = None
        self.start_time = datetime.utcnow() # Reference time (t=0) for simulation offsets

    def start_matlab_engine(self):
        """Start the MATLAB engine and change directory."""
        st.info("Starting MATLAB engine...")
        try:
            self.eng = matlab.engine.start_matlab()
            self.eng.cd(self.matlab_script_path, nargout=0)
            st.success("MATLAB engine started.")
            return True
        except Exception as e:
            st.error(f"Failed to start MATLAB engine or change directory: {e}")
```

```
            st.info(f"Ensure MATLAB is running and the path '{self.matlab_script_path}' is correct.")
            self.eng = None
            return False

    def run_simulation(self):
        """Run the Simulink function via MATLAB engine."""
        if self.eng is None:
            if not self.start_matlab_engine():
                return False # Indicate failure

        model_name = "Switch"
        condition_value = float(self.condition)

        st.info(f"Running MATLAB simulation: Model='{model_name}', Condition={condition_value}...")
        try:
            raw_output = self.eng.Run_Simulink_Function(model_name, condition_value, nargout=1)
            self.function_output = np.array(raw_output)

            if self.function_output is None or self.function_output.size == 0:
                st.error("MATLAB function returned no output.")
                self.function_output = None
                return False

            if len(self.function_output) > self.max_iterations:
                st.warning(f"Output truncated to {self.max_iterations} iterations.")
                self.function_output = self.function_output[:self.max_iterations]

            st.success(f"MATLAB simulation done. Output shape: {self.function_output.shape}")
            return True

        except matlab.engine.MatlabExecutionError as me:
            st.error(f"MATLAB Execution Error: {me}")
            st.error("Check the MATLAB script ('Run_Simulink_Function.m') and model ('Switch.slx') for issues.")
            self.function_output = None
            return False
        except Exception as e:
            st.error(f"Error during MATLAB simulation run: {e}")
            self.function_output = None
            return False

    # ------------------------------------------------------------
    # CORRECTED TIMESTAMP FUNCTION
    # ------------------------------------------------------------
    def matlab_datenum_to_iso(self, simtime):
        """Convert MATLAB simulation time offset to ISO format timestamp."""
```

```
89          try:
90              # Convert simtime (offset in seconds from start) to float
91              simtime_float = float(simtime)
92              # Calculate the absolute time for this data point
93              actual_time = self.start_time + timedelta(seconds=
    simtime_float)
94
95              # Generate ISO format string.
96              # REMOVED timespec='nanoseconds' which caused 'Unknown
    timespec' error on Python < 3.11
97              # 'auto' (default) includes microseconds if they are non-zero.
98              iso_timestamp = actual_time.isoformat(timespec='microseconds')
     # Explicitly request microseconds
99
100             # Ensure the timestamp ends with 'Z' to denote UTC timezone
    for InfluxDB
101             # self.start_time is UTC, so actual_time is also UTC
102             if not iso_timestamp.endswith('Z'):
103                 # Check if timezone info is already present (e.g.,
    +00:00)
104                 # This shouldn't happen with utcnow() but check just in
    case.
105                 if '+' not in iso_timestamp and '-' not in iso_timestamp
    [10:]: # Avoid matching date hyphen
106                     iso_timestamp += 'Z'
107
108             return iso_timestamp
109         except Exception as e:
110             # Display error in Streamlit UI for debugging
111             st.warning(f"Timestamp conversion error for simtime {repr(
    simtime)}: {e}")
112             return None
113     # ------------------------------------------------------------------
114
115     def write_to_influxdb(self, measurement_name, fields_mapping):
116         """Write simulation output data to InfluxDB."""
117         if self.function_output is None or len(self.function_output) == 0:
118             st.warning(f"No simulation data available to write for {
    measurement_name}.")
119             return False
120
121         try:
122             if not self.client.ping():
123                 st.error(f"InfluxDB connection failed (ping) for writing
    to {measurement_name}.")
124                 return False
125         except Exception as ping_e:
126              st.error(f"InfluxDB connection error during ping: {ping_e}")
127              return False
128
129         write_api = self.client.write_api(write_options=SYNCHRONOUS)
130         num_rows = len(self.function_output)
131         points_to_write = []
132         rows_skipped_ts = 0
```

```
133         rows_skipped_field = 0
134         rows_skipped_nan = 0
135 
136         st.info(f"Preparing {num_rows} data points for InfluxDB measurement '{measurement_name}'...")
137 
138         for i in range(num_rows):
139             row_data = self.function_output[i]
140             simtime = row_data[0] # First element is time
141 
142             # Use the corrected conversion function
143             simtime_iso = self.matlab_datenum_to_iso(simtime)
144             if simtime_iso is None:
145                 rows_skipped_ts += 1
146                 continue # Skip row if timestamp is invalid
147 
148             p = influxdb_client.Point(measurement_name).tag("Machine_Type", str(self.condition))
149             valid_field_added = False
150             for field, index in fields_mapping.items():
151                 try:
152                     value = float(row_data[index])
153                     if not np.isnan(value) and not np.isinf(value):
154                         p = p.field(field, value)
155                         valid_field_added = True
156                     else:
157                         rows_skipped_nan += 1
158                 except IndexError:
159                     rows_skipped_field += 1
160                     if f"field_idx_warn_{field}_{measurement_name}" not in st.session_state:
161                         st.warning(f"Measurement '{measurement_name}': Index {index} out of bounds for field '{field}'.")
162                         st.session_state[f"field_idx_warn_{field}_{measurement_name}"] = True
163                     valid_field_added = False
164                     break
165                 except Exception as field_e:
166                     st.warning(f"Measurement '{measurement_name}', Row {i}: Error processing field '{field}': {field_e}")
167 
168             if valid_field_added:
169                 # Still tell the client library the precision is nanoseconds,
170                 # even if the ISO string only shows microseconds. InfluxDB stores in nanoseconds.
171                 p = p.time(simtime_iso, write_precision='ns')
172                 points_to_write.append(p)
173 
174         if rows_skipped_ts > 0: st.warning(f"Skipped {rows_skipped_ts} rows due to timestamp errors for {measurement_name}.")
175         if rows_skipped_field > 0: st.warning(f"Skipped fields/rows due to index errors for {measurement_name}.")
176         if rows_skipped_nan > 0: st.warning(f"Skipped {rows_skipped_nan}
```

```
    NaN/Inf field values for {measurement_name}.")

        if not points_to_write:
            st.warning(f"No valid data points generated to write for {measurement_name}.")
            return False

        st.info(f"Writing {len(points_to_write)} points to InfluxDB '{measurement_name}'...")
        try:
            write_api.write(bucket=self.bucket, org=self.org, record=points_to_write)
            st.success(f"Successfully wrote {len(points_to_write)} points to {measurement_name}.")
            return True
        except Exception as e:
            st.error(f"Error writing to InfluxDB {measurement_name}: {e}")
            return False

    def stop_engine(self):
        """ Explicitly stops the MATLAB engine if running. """
        if self.eng:
            try:
                self.eng.quit()
                st.info("MATLAB engine stopped.")
            except Exception as e:
                st.warning(f"Could not stop MATLAB engine cleanly: {e}")
            self.eng = None


# (InfluxDBQuery and DataVisualization classes remain the same as previous version)
class InfluxDBQuery:
    """ Handles querying data from InfluxDB. """
    def __init__(self, url, token, org, bucket):
        self.org = org
        self.bucket = bucket
        self.client = influxdb_client.InfluxDBClient(url=url, token=token, org=org, verify_ssl=False)

    def query_data(self, measurement, start_time_iso, stop_time_iso):
        """ Query data from InfluxDB using Flux (takes ISO strings). """
        query = f"""
        from(bucket: "{self.bucket}")
            |> range(start: {start_time_iso}, stop: {stop_time_iso})
            |> filter(fn: (r) => r._measurement == "{measurement}")
            |> yield(name: "results")
        """
        st.info(f"Querying InfluxDB: Measurement='{measurement}', Range='{start_time_iso}' to '{stop_time_iso}'")
        try:
            query_api = self.client.query_api()
            result = query_api.query(query=query, org=self.org)
            # st.success("InfluxDB query successful.") # Less verbose
```

```
            return result
        except Exception as e:
            st.error(f"Error executing InfluxDB query: {e}")
            return None

    def process_records(self, query_result):
        """ Process Flux query result into a Pandas DataFrame. """
        if query_result is None: return pd.DataFrame()
        # st.info("Processing query results into DataFrame...") # Less verbose
        records = []
        try:
            for table in query_result:
                for record in table.records:
                    records.append({
                        "timestamp": record.get_time(),
                        "Machine_Type": record.values.get("Machine_Type", "N/A"),
                        "_field": record.get_field(),
                        "value": record.get_value()
                    })
            if not records:
                # st.warning("Query returned no records.") # Less verbose
                return pd.DataFrame()
            df = pd.DataFrame(records)
            df['timestamp'] = pd.to_datetime(df['timestamp'])
            # st.success(f"DataFrame created with {len(df)} records.") # Less verbose
            return df
        except Exception as e:
            st.error(f"Error processing query results: {e}")
            return pd.DataFrame()


class DataVisualization:
    """ Handles plotting of the queried InfluxDB data. """
    def __init__(self, df):
        if not isinstance(df, pd.DataFrame) or df.empty:
             self.df = pd.DataFrame()
             # st.info("No data available for visualization.") # Less verbose
        else:
             self.df = df

    def display_data(self, selected_field):
        """ Displays time series plot """
        if self.df.empty: return
        if selected_field not in self.df['_field'].unique():
            st.warning(f"Field '{selected_field}' not found in the queried data.")
            return
        df_filtered = self.df[self.df['_field'] == selected_field].copy()
        if df_filtered.empty:
            st.info(f"No data points found for field '{selected_field}'.")
```

```
            return
        df_filtered['timestamp'] = pd.to_datetime(df_filtered['timestamp'])
        df_filtered.sort_values('timestamp', inplace=True)
        st.subheader(f"Time Series Plot for {selected_field}")
        st.line_chart(df_filtered.set_index('timestamp')['value'])

    def display_pivot_table(self):
        """ Displays a pivot table view of the data. """
        st.subheader("Pivoted Data View")
        if self.df.empty:
            st.info("No data to display in table.")
            return
        try:
            df_pivoted = self.df.pivot_table(
                index=["timestamp", "Machine_Type"], columns="_field", values="value", aggfunc="first"
            )
            st.dataframe(df_pivoted)
        except Exception as e:
            st.error(f"Could not create pivot table: {e}")


# --- Streamlit App Structure ---
st.set_page_config(page_title="Simulation & Viz", layout="wide")
st.title("       Load Selection & Visualization of Energy Consumption")

# --- Globals/Configuration ---
DEFAULT_MATLAB_SCRIPT_PATH = '/Users/rishikeshtiwari/THK/Case Studies/Case Study IOT'
INFLUX_URL = "https://eu-central-1-1.aws.cloud2.influxdata.com"
INFLUX_TOKEN_SIM = "Jzj8SoxpTpDq5FM0JPoYx93wlmJC7BJ7pqYV9GwJMrdp_33DemOXh7SoTIQaLgBfKqnqgdtUAiBilEgd0ck=="
INFLUX_TOKEN_VIZ = "ZKu1zQnBhbh1_ywDqH6nQqlVapG_3Nv8v3gA9u4dD_isRGTUtWHprjJwACMO3oG7y2hPkxRRhPvUy23LuUO=="
INFLUX_ORG = "Student"
INFLUX_BUCKET = "CPEMS"

machine_options = {"Motor": 1, "Induction Furnace": 2, "General Electric Load": 4, "Pump": 5}
condition_mapping = {
    (1,): 1, (2,): 2, (1, 2): 3, (4,): 4, (5,): 5, (4, 5): 6, (1, 4): 7,
    (1, 5): 8, (4, 2): 9, (2, 5): 10, (1, 4, 5): 11, (2, 5, 1): 12,
    (4, 2, 1): 13, (2, 4, 5): 14, (1, 2, 4, 5): 15
}
fields_map_siemens = {
    "voltage u1": 1, "voltage u2": 2, "voltage u3": 3, "current i1": 4, "current i2": 5,
    "current i3": 6, "active_power p1": 7, "active_power p2": 8, "active_power p3": 9,
    "reactive_power q1": 10, "reactive_power q2": 11, "reactive_power q3": 12,
```

```
    "apparent_power s1": 13, "apparent_power s2": 14, "apparent_power s3": 15,
    "power_factor pf1": 16, "power_factor pf2": 17, "power_factor pf3": 18
}
fields_map_beckhoff = {"voltage u1": 1, "voltage u2": 2, "voltage u3": 3, "current i1": 4, "current i2": 5, "current i3": 6}

# --- Tabs ---
sim_tab, viz_tab = st.tabs(["MATLAB Simulation Control", "Data Visualization"])

# --- Simulation Tab ---
with sim_tab:
    st.header("Run Simulation and Upload")

    # Display Python version for debugging purposes
    st.caption(f"Running on Python {sys.version}")

    if not MATLAB_AVAILABLE:
        st.error(
            "MATLAB Engine for Python not installed or MATLAB not found.\n"
            "Simulation features requiring MATLAB will be disabled.\n\n"
            "**To enable:** Check MATLAB installation and Python engine setup."
        )
    else:
        matlab_script_path_sim = st.text_input(
            "MATLAB Script Directory Path", DEFAULT_MATLAB_SCRIPT_PATH, key="sim_path_input"
        )
        selected_machines_sim = st.multiselect(
            "Select Machine Types for Simulation:", list(machine_options.keys()), key="sim_machine_select"
        )
        selected_values_sim = tuple(sorted(machine_options[m] for m in selected_machines_sim))
        condition_sim = condition_mapping.get(selected_values_sim, 0)
        st.write(f"Selected Condition Number: **{condition_sim}**")
        max_iterations_sim = st.number_input(
            "Max Simulation Iterations to Process", min_value=100, value=5000, step=100, key="sim_max_iter"
        )

        if st.button("       Run MATLAB Simulation & Upload Data", key="run_sim_button"):
            run_simulation_flag = True
            if not selected_machines_sim:
                st.warning("Please select at least one machine type.")
                run_simulation_flag = False
            if condition_sim == 0 and selected_machines_sim:
                st.error("Selected machine combination is not mapped to a valid condition number.")
                run_simulation_flag = False
```

```
            if not os.path.isdir(matlab_script_path_sim):
                st.error(f"Invalid MATLAB script directory path: '{matlab_script_path_sim}'.")
                run_simulation_flag = False

            if run_simulation_flag:
                loader = MatlabDataLoader(
                    condition_sim, matlab_script_path_sim, INFLUX_URL, INFLUX_TOKEN_SIM,
                    INFLUX_ORG, INFLUX_BUCKET, max_iterations_sim
                )
                sim_success = loader.run_simulation()
                if sim_success and loader.function_output is not None:
                    st.info("Simulation successful. Proceeding to data upload...")
                    success1 = loader.write_to_influxdb("Siemens_1", fields_map_siemens)
                    success2 = loader.write_to_influxdb("Beckhoff_1", fields_map_beckhoff)
                    if success1 or success2: st.success("Data upload process completed.")
                    else: st.error("Data upload failed or no valid points written.")
                else:
                    st.error("Simulation failed or produced no data. Cannot upload.")
                loader.stop_engine()

# --- Visualization Tab ---
with viz_tab:
    st.header("Visualize InfluxDB Data")
    influx_querier = InfluxDBQuery(INFLUX_URL, INFLUX_TOKEN_VIZ, INFLUX_ORG, INFLUX_BUCKET)
    viz_query_col1, viz_query_col2 = st.columns(2)
    with viz_query_col1:
        measurements_options = ["Beckhoff_1", "Siemens_1", "Rough_6"]
        selected_measurement_viz = st.selectbox( "Select Measurement", measurements_options, key="viz_measurement_select")
        default_start_datetime = datetime.now() - timedelta(days=1)
        start_date_viz = st.date_input("Start Date", default_start_datetime.date(), key="viz_start_date")
        start_time_viz = st.time_input("Start Time", time(0, 0), key="viz_start_time")
    with viz_query_col2:
        default_end_datetime = datetime.now()
        stop_date_viz = st.date_input("Stop Date", default_end_datetime.date(), key="viz_stop_date")
        stop_time_viz = st.time_input("Stop Time", time(23, 59, 59), key="viz_stop_time")

    start_datetime_viz = datetime.combine(start_date_viz, start_time_viz)
    stop_datetime_viz = datetime.combine(stop_date_viz, stop_time_viz)
    start_time_iso = start_datetime_viz.isoformat(timespec='seconds') + 'Z'
```

```
    stop_time_iso = stop_datetime_viz.isoformat(timespec='seconds') + 'Z'

    if st.button("     Query InfluxDB Data", key="query_influx_button"):
        query_result = influx_querier.query_data(selected_measurement_viz, start_time_iso, stop_time_iso)
        df_viz = influx_querier.process_records(query_result)
        st.session_state['viz_dataframe'] = df_viz
        st.session_state['viz_measurement'] = selected_measurement_viz
        st.session_state['viz_fields'] = sorted(df_viz['_field'].unique()) if not df_viz.empty else []
        # Clear selection if fields change - prevents errors if old field disappears
        if 'viz_field_select' in st.session_state:
            current_selection = st.session_state['viz_field_select']
            if current_selection not in st.session_state['viz_fields']:
                del st.session_state['viz_field_select']


    if 'viz_dataframe' in st.session_state and isinstance(st.session_state['viz_dataframe'], pd.DataFrame):
        df_display = st.session_state['viz_dataframe']
        if not df_display.empty:
            st.markdown("---")
            st.subheader(f"Visualization for Measurement: {st.session_state.get('viz_measurement', 'N/A')}")
            available_fields = st.session_state.get('viz_fields', [])
            if available_fields:
                # Use session state key directly in selectbox to preserve selection if possible
                selected_field_viz = st.selectbox(
                   "Select Field to Plot", options=available_fields, key="viz_field_select"
                )
                show_pivot_data = st.checkbox('Show Data Table (Pivoted View)', key="viz_show_pivot", value=False)
                try:
                   data_viz = DataVisualization(df_display)
                   data_viz.display_data(selected_field_viz)
                   if show_pivot_data: data_viz.display_pivot_table()
                except Exception as viz_e:
                   st.error(f"An error occurred during visualization: {viz_e}")
            else:
                st.info("Query returned data, but no specific fields were found.")
        elif 'viz_measurement' in st.session_state:
            st.info(f"No data found for measurement '{st.session_state['viz_measurement']}' in the selected time range.")
```

Listing 3: Streamlit Simulation Tab Code

A.2 Matlab Main Function Code

```
function [a] = Run_Simulink_Function(b, c)
    mdl = b;
    x = c;
    % Open and configure the Simulink model
    open_system(mdl);
    % Set the value of the Constant Block (assuming block path is 'mdl/
   ConstantBlockName')
    set_param([mdl, '/Constant'], 'Value', num2str(x)); % A_value is the
   value you want to assign
    % Configure simulation parameters
    simIn = Simulink.SimulationInput(mdl);
    simIn = setModelParameter(simIn, 'StopTime', '5');
    simIn = setModelParameter(simIn, 'ZeroCrossControl', 'disable');
    % Run the simulation
    out = sim(simIn);
    % Extract simulation results
    outputV = out.V1ind;
    timecolumnV = outputV.Time;
    voltage1_data = out.V1ind.Data;
    current1_data = out.I1ind.Data;
    active1_data = out.P1.Data;
    reactive1_data = out.Q1.Data;
    apparent1_data = out.A1.Data;
    PF1_data = out.PF1.Data;
    voltage2_data = out.V2ind.Data;
    current2_data = out.I2ind.Data;
    active2_data = out.P2.Data;
    reactive2_data = out.Q2.Data;
    apparent2_data = out.A2.Data;
    PF2_data = out.PF2.Data;
    voltage3_data = out.V3ind.Data;
    current3_data = out.I3ind.Data;
    active3_data = out.P3.Data;
    reactive3_data = out.Q3.Data;
    apparent3_data = out.A3.Data;
    PF3_data = out.PF3.Data;
    % Create the output matrix
    OutMatrix = [timecolumnV voltage1_data voltage2_data voltage3_data ...
                 current1_data current2_data current3_data ...
                 active1_data active2_data active3_data ...
                 reactive1_data reactive2_data reactive3_data ...
                 apparent1_data apparent2_data apparent3_data ...
                 PF1_data PF2_data PF3_data];
    a = OutMatrix;
end
```

Listing 4: MATLAB Function for Grid Simulation

A.3 Multi-Output-Switch Code

```
function [output1, output2, output3, output4] = multiOutputSwitch(
    condition)

    % Initialize the outputs to specific values (e.g., 0 or NaN) for
    Simulink to infer size
    output1 = NaN;  %     Fixed: NaN (uppercase N)
    output2 = NaN;  %     Fixed: NaN (uppercase N)
    output3 = NaN;  %     Fixed: NaN (uppercase N)
    output4 = NaN;  %     Fixed: NaN (uppercase N)

    % Check the condition and assign outputs
    switch condition
        case 1
            output1 = 1;
        case 2
            output2 = 1;
        case 3
            output1 = 1;
            output2 = 1;
        case 4
            output3 = 1;
        case 5
            output4 = 1;
        case 6
            output3 = 1;
            output4 = 1;
        case 7
            output1 = 1;
            output3 = 1;
        case 8
            output1 = 1;
            output4 = 1;
        case 9
            output3 = 1;
            output2 = 1;
        case 10
            output2 = 1;
            output4 = 1;
        case 11
            output3 = 1;
            output4 = 1;
            output1 = 1;
        case 12
            output2 = 1;
            output4 = 1;
            output1 = 1;
        case 13
            output3 = 1;
            output2 = 1;
            output1 = 1;
        case 14
            output2 = 1;
```

```
            output4 = 1;
            output3 = 1;
        case 15
            output3 = 1;
            output4 = 1;
            output2 = 1;
            output1 = 1;
        otherwise
            % Handle other conditions or keep the default (NaN)
    initialization
            output1 = NaN;
            output2 = NaN;
            output3 = NaN;
            output4 = NaN;
     end
end
```

Listing 5: MATLAB function to handle multiple outputs based on a switch condition