Integrating Ecovisor into Mosaik Co-Simulation

Marvin Steinke and Henrik Nickel
Technische Universität Berlin

***Abstract*—To reduce emissions, cloud platforms must increasingly rely on renewable energy sources such as solar and wind power. Nevertheless, the issue of volatility associated with these sources presents a significant challenge, since current energy systems conceal such unreliability in hardware. Souza et al. have devised a solution to this issue by creating an "ecovisor". This system virtualizes the energy infrastructure and allows for software-defined control to be accessible by applications. Setting up the ecovisor to develop carbon-aware applications, however, can be costly and time consuming. To address this problem, we simulated the ecovisor and its virtual energy system and integrated in into a Mosaik co-simulation. With an API server and a Redis database we are enabling Software- (SIL) and Hardware-In-The-Loop (HIL) capabilities. To evaluate our approach, we created test cases using recorded solar and carbon data to demonstrate the accuracy of the ecovisor model's implementation and its ability to transfer data correctly between the model and the API server.**

I. Introduction

THE surge of cloud platforms has had a significant impact on many businesses and individuals, offering access to innovative and valuable applications that frequently require significant computational resources. A notable example for this is represented by OpenAI's chat generative pre-trained transformer (ChatGPT) cloud platform, which is based on the GPT-3 model developed by Brown et al. [1] and achieved more than a million users within the first five days of its launch [2]. The increasing demand for more computational power has lead cloud platforms to become an essential part of the digital landscape [3]. These platforms allow for the storage, processing and management of large amounts of data and computational resources, which can be used to run applications and services that are beyond the capabilities of traditional hardware systems.

However, while the growth of cloud platforms has brought many benefits, it has also raised concerns about their environmental impact. As the demand for computational resources increases, so does the carbon emissions generated by the energy consumption of these platforms [4]. Despite their ecological impact, the growth of cloud platforms shows no signs of slowing down. According to Gartner Inc. worldwide end-user spending on public cloud services is forecasted to grow 20.7% to total $591.8 billion in 2023 [5]. To mitigate their impact on the environment, cloud platforms are now looking for ways to reduce their carbon footprint [6, 7]. It is imperative to adopt cleaner energy sources for powering data centers, both in the cloud and at the edge.

Although clean energy offers numerous advantages, it is perceived as being unreliable due to two key factors. One, the generation of renewable energy sources such as solar and wind is affected by environmental changes, and two, the carbon-intensity of grid power fluctuates as the grid uses different types of generators with varying carbon emissions to meet demand [8]. The field of computing possesses a distinct advantage in terms of reducing its carbon impact through the utilization of cleaner energy sources [9]. However, current cloud applications are unable to apply these benefits to optimize their carbon efficiency. This is because the energy system obscures the instability of clean energy with a reliability abstraction, which gives no control or visibility into the energy supply. As a result, applications cannot adjust their power usage in response to changes in the availability and carbon-intensity of renewable energy [10].

Souza et al. [10] proposed a solution to this issue by creating an "ecovisor", which virtualises the energy system and provides software-based control over it. This ecovisor enables each application to manage the instability of clean energy through software, customized to meet its unique requirements. However, even their small-scale prototype is intricately designed and incorporates several expensive components, such as a DC power supply equipped with a solar array simulation capability that costs almost $10,000.

Though the creation of the ecovisor is a promising solution to the issue of reducing the carbon footprint of cloud platforms, a large scale prototype for research and development purposes would not only be significantly cost intensive, but also time consuming. An alternative approach to consider is to simulate *only* the ecovisor infrastructure with real applications as a Software-In-The-Loop (SIL) implementation. This would allow for the optimization of energy usage in response to changes in the availability and carbon-intensity of renewable energy, without the need for a physical implementation. By simulating the ecovisor, applications can still manage the instability of clean energy through software, but at a lower cost and with less time investment.

To this end, we propose the utilization of a co-simulation tool, Mosaik, to integrate this ecovisor simulation into, and evaluate its impact on the carbon footprint of cloud platforms. Mosaik is a simulation framework for power systems, communication networks, and building automation, which can be used to model and analyze the performance of the ecovisor in real-world applications [11].

In the following sections of this article, we will commence by presenting the necessary contextual information in Section II to ensure comprehension of our methodology. This includes co-simulation environments with Software- (SIL) and Hardware-In-The-Loop (HIL) strategies, the Mosaik co-simulation tool and a closer examination of the ecovisor infrastructure and its interface to applications. We will then review some related literature with similar approaches in Section

III. Subsequently, we present our approach of integrating the ecovisor into Mosaik in Section IV. The approach is divided into the simulation of the ecovisor itself and the interface to external applications, beyond the scope of the co-simulation. We evaluate our approach in Section V with exemplary data, representing a realistic scenario that covers all edge cases. After discussing future research directions related to this article in Section VI, we conclude this article in Section VII by summarizing the main points and contributions.

II. Background

A. Co-Simulations and SIL / HIL

Co-simulations are used to model and analyze complex systems with multiple interacting components, each of which may have different properties and behaviors. They involve combining simulation models from different domains, such as control, power, and thermal management, to create a unified model that accurately represents the behavior of the overall system.

One of the main advantages of co-simulations over regular simulations is their ability to capture the interactions between different components of the system. Regular simulations often make simplifying assumptions that can lead to inaccurate results. Co-simulations, on the other hand, can account for the interactions between components and provide a more accurate representation of the system's behavior. This makes co-simulations particularly useful for designing and optimizing complex systems like datacenters. The virtual environment can save time and resources, reduce the risk of failure, and lead to more efficient and sustainable datacenters [12].

Co-simulations can be further enhanced by incorporating the SIL and HIL methodology. This approach involves integrating real-world components, such as software and hardware, into the simulation environment to better reflect the actual system behavior. The integration of real components allows for a more accurate representation of the system's behavior and can also identify potential issues that may arise in real-world scenarios [13].

B. Mosaik

Mosaik is an open-source co-simulation tool that allows for the integration of different simulation models from various domains into a unified simulation environment [11]. Mosaik's four main components enable communication between simulators and Mosaik, the creation of simulation scenarios, the management of simulator processes, and the coordinated simulation of a scenario.[1]

Mosaik Sim API. The Mosaik Sim API provides a standardized communication protocol for simulators and Mosaik to exchange information. It utilizes plain network sockets and JSON encoded messages to facilitate communication between simulators and Mosaik. Although the API provides a low-level communication protocol, it is complemented by a high-level API for some programming languages. The high-level API abstracts the networking-related aspects of the communication protocol, allowing users to focus on the core logic of their simulation models. To use the high-level API, users only need to write a subclass and implement a few required methods.

Scenario API. This empowers users to create simulation scenarios entirely in Python. This API enables users to launch simulators, create model instances, and generate entity sets that can be connected to establish data flows between different simulators. With the Scenario API, users can easily connect individual entities or entire sets of entities to achieve their simulation goals.

SimManager. The Simulator Manager, or SimManager, handles simulator processes and their communication with Mosaik. The SimManager can start new simulator processes, connect to already running process instances, and import a simulator module and execute it in-process if it is written in Python 3. In-process execution reduces memory requirements and avoids the overhead of serializing and sending messages over the network. External processes, however, can be executed in parallel, which is not possible with in-process simulators.

Mosaik's Scheduler. Mosaik's Scheduler uses the event-discrete simulation library SimPy for the coordinated simulation of a scenario. Mosaik supports time-discrete and event-discrete simulations, as well as a combination of both paradigms. It can handle simulators with different step sizes, and a simulator may vary its step size during the simulation. Mosaik tracks the dependencies between simulators and only lets them perform a simulation step if necessary. It is also able to let multiple simulators perform their simulation step in parallel if they do not depend on each other's data.

C. Ecovisor [2]

Figure 1 shows an overview of the ecovisor's design which manages resources and energy using containers or virtual machines as the basic unit. An instance-level API is chosen to align with existing cloud APIs and to support higher-level cluster or cloud-level APIs. The ecovisor extends an existing orchestration platform that provides basic container or VM management and monitoring functions. Container Orchestration Platforms (COPs) are used to manage resources and applications. COPs provide virtual clusters composed of multiple containers with specified resource allocations that can grow or shrink over time. COPs include a scheduling policy that determines resource allocation under constraints, and COPs are resilient to resource revocations. This resiliency is useful for designing carbon-efficient applications as low-carbon energy may cause power shortages that also manifest as resource revocations.

The virtual energy system includes a virtual grid connection, a virtual battery, and a virtual solar array. The system provides getters and setters methods for monitoring and controlling

[1]The information about the main components is adapted from the official documentation [14].

[2]The information presented in this section is a summary of Section 3 and 3.1 from Souza et al.'s work [10], with some modifications made for clarity and conciseness.

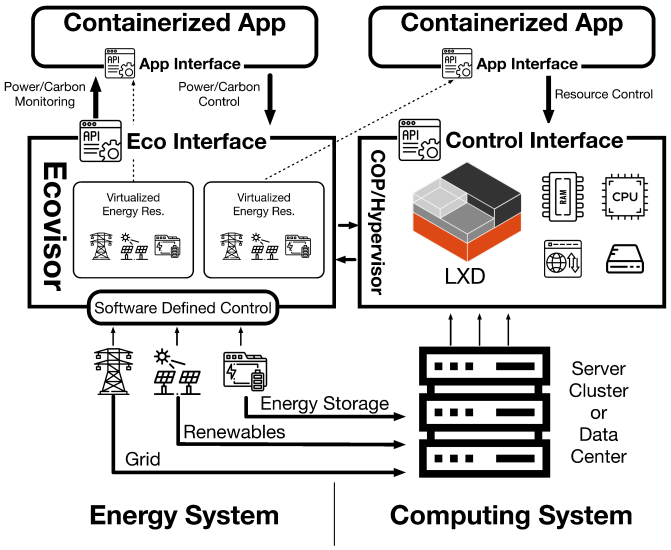

Figure 1. Ecovisor Design (Souza et al.) [10]

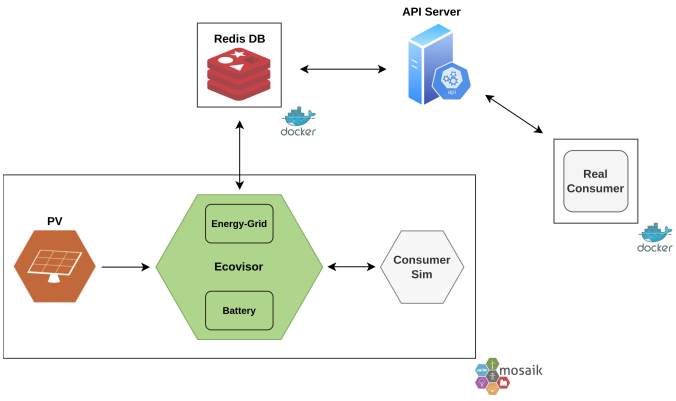

Figure 2. General System Design: The ecovisor infrastructure is simulated and integrated into the Mosaik co-simulation environment. SIL capabilities are enabled via an API Server and a Redis database, connecting the simulation environment and real applications in real-time.

the virtual power supply and demand, including per-container power caps, battery charging, and discharging rates, as shown in Table I. The system uses virtual solar power first to meet demand and charges the virtual battery with any excess solar power. When there is not enough solar power, the virtual energy system uses grid power to make up the difference, while accounting for carbon emissions and power usage over discrete time intervals. The ecovisor system provides a uniform centralized interface for accessing energy-related information and historical data.

III. Related Work

In research and development processes, the utilization of physical testbeds is a widespread practice [15, 16]. They serve as a medium for testing and evaluating new technologies, systems, and products prior to their market launch or further development. However, some scenarios and conditions may not be feasible or safe to recreate in a physical testbed. Simulations can provide a controlled and cost-effective environment for testing such scenarios and conditions [17]. When specific physical or software components require a certain degree of realism though, simulations or co-simulations with SIL or HIL methodologies can effectively address these limitations.

For instance, Beilharz et al. [18], introduce Marvis, a framework that provides a comprehensive staging environment for testing distributed IoT applications. Marvis orchestrates hybrid testbeds, co-simulated domain environments, and a central network simulation to create a representative operating environment for the system. However, Marvis does not provide a virtualized energy system, which is crucial to meet the requirements of our problem statement.

Hagenmeyer et al. [19] investigate the interplay of different forms of energy on various value chains in Energy Lab 2.0. The focus is on finding novel concepts to stabilize the volatile energy supply of renewables through the use of storage systems and the application of information and communication technology tools and algorithms. The smart energy system simulation and control center is a key element of Energy Lab 2.0 and consists of three parts: a power-hardware-in-the-loop experimental field, an energy grid simulation and analysis laboratory, and a control, monitoring, and visualization center. While this smart energy system simulation is similar to the simulated ecovisor in our approach, the control center is the only entity with software-based control over this energy system. The ecovisor infrastructure, however, provides multiple applications with the ability to manage their energy supply themselves which is an essential factor for our approach.

In conclusion, to the best of our knowledge, there is no current approach that can simulate an energy system virtualizer with software-based control, comparable to the ecovisor, with SIL capabilities.

IV. Approach

Figure 2 illustrates our general system design approach that simulates the ecovisor infrastructure and integrates it into the Mosaik co-simulation environment while enabling SIL capabilities. The present design can be categorized into two distinct parts, the simulation of the ecovisor, and its interface to external applications, which are elaborated on in the following.

A. Simulation of the Ecovisor

The simulation part of our approach is fully contained within the Mosaik co-simulation environment. In Figure 2 this includes the photo-voltaic (PV) module, the ecovisor and a consumer simulator. The consumer and PV module simulators utilized in this context are the `mosaik-csv` simulator, which is a component of the Mosaik ecosystem [20]. This simulator is capable of simulating CSV datasets, and in the current application, it is employed to simulate consumption data and recorded or forecasted PV data. While our primary focus is to facilitate the integration of external, non-simulated applications with our simulated ecovisor, we also recognize the need to accommodate simulated consumers for more sophisticated approaches that require additional data. Therefore, we aim to provide flexibility in our system to meet a wide range of requirements. Because these two components are designed to

Table I
ECOVISOR'S API (SOUZA ET AL.) [10]

Function Name	Type	Input	Return Value	Description
`set_container_powercap()`	Setter	ContainerID, kW	N/A	Set a container's power cap
`set_battery_charge_rate()`	Setter	kW	N/A	Set battery charge rate until full
`set_battery_max_discharge()`	Setter	kW	N/A	Set max battery discharge rate
`get_solar_power()`	Getter	N/A	kW	Get virtual solar power output
`get_grid_power()`	Getter	N/A	kW	Get virtual grid power usage
`get_grid_carbon()`	Getter	N/A	$g \cdot CO_2/kW$	Get virtual grid power usage
`get_battery_discharge_rate()`	Getter	N/A	kW	Get current rate of battery discharge
`get_battery_charge_level()`	Getter	N/A	kWh	Get energy stored in virtual battery
`get_container_powercap()`	Getter	ContainerID	kW	Get a container's power cap
`get_container_power()`	Getter	ContainerID	kW	Get a container's power usage
`tick()`	Notification	N/A	N/A	Invoked by ecovisor every Δt

be minimal and replaceable, they each require a controller agent that acts as a medium between the component and the simulated ecovisor. The controller can be initialized with e.g. a power conversion factor to allow for the use of different measurement units in the datasets, as the standard power unit utilized in the ecovisor is kW.

The simulated ecovisor holds all necessary variables to enable the API in Table I. This includes instances of a simple battery and energy-grid model. The Simple Battery model can be configured with a capacity and an initial charge. For every simulation step, the charge is incremented by the delta value. The Simple Energy-Grid model reads carbon information from a CSV dataset, either every simulation step line by line, or for a specific point in time.

Because the co-simulation runs in real or "wall-clock" time, the Mosaik scenario needs to be executed with `rt_factor=1`. Most models in our approach are time-based and one simulation step always translates to one second in real-time in our environment. Therefore, we only model power as energy, making the standard unit for our approach kWs instead of kW.

The simulation of the ecovisor's virtual energy system is realized in Algorithm 1. Like its original counterpart, the system prioritizes the utilization of virtual solar power to fulfill energy requirements in line 1. The power values are centered around consumption. Because solar power is *adding* power to the system, its value is negative. Therefore, if `rest` is negative or zero in line 2, there is extra solar power and the battery does not discharge in line 3. In cases where solar power is insufficient, the virtual energy system resorts to battery to compensate for the shortfall in lines 5 – 9. The discharge rate of the battery then depends on its maximum discharge rate, the charge level of the battery and the power that is attempted to be drawn. If the battery can not supply sufficient power, the remaining power is being drawn from the grid in line 12. Lines 13 – 15 then adjust the battery's charge. Finally lines 16 – 18 determine the carbon concentration for the simulation step.

Our approach does not implement a COP/Hypervisor. In the ecovisor, the COP serves the primary purpose of granting containerized applications control over their power consumption. Table I shows that each container can use the `set_container_powercap()` function to set their power consumption limit. The ecovisor measures the power consumption of the container using PowerAPI [21] and applies limits on resource utilization using cgroups. The issue with PowerAPI is that it relies on dedicated hardware equipped with sensors to collect raw data on software power consumption. This collected data is then processed by a computational module that utilizes a formula to *estimate* the power consumption. A key motivation behind the integration of the ecovisor into a co-simulation is to create an affordable and accessible platform for researching and developing carbon-aware applications. As power consumption measurement and control can vary greatly depending on the research and development environment, we have chosen to avoid implementing a specific strategy in order to allow for greater flexibility and freedom.

```
1  rest ← consumption − solar
2  if rest ≤ 0 then
       // excess (or equal) solar power
3      b_discharge_rate ← 0
4  else
       // solar power is insufficient
           → use battery
5      b_discharge_rate ← min(
6          b_max_discharge,
7          b_charge_level · 3600,
8          rest
9      )
10     rest ← rest − b_discharge_rate
11 end
   // draw remaining power from grid
12 grid_power ← b_charge_rate + rest
   // adjust battery charge
13 b.delta ← b_charge_rate − b_discharge_rate
14 b.step()
15 b_charge_level ← b.charge
   // determine carbon concentration for
       step
16 energy_grid.step()
17 grid_carbon ← energy_grid.carbon
18 total_carbon ← grid_carbon · grid_power
```

Algorithm 1: Virtual Energy System Simulation

B. Interface to External Applications

To integrate the ecovisor model with a real workload application, we exposed the API as described in Table I to containerized workloads. This API was then integrated into a FastAPI server [22]. The API server is connected to a Redis database [23], which acts as a key-value store and establishes a link between the API server and the ecovisor model. The ecovisor model itself has been designed to implement a Redis interface that enables it to read and write data from the RedisDB. In the following sections, we will provide a detailed description of the three components that constitute the External Application Interface.

1) API-Server: The ecovisor API is exposed to the workloads via the API-Server, which is implemented using the FastAPI Framework and utilizes the Uvicorn web server [24] to handle multiple clients accessing the API. Since the web server uses the ASGI (Asynchronous Server Gateway Interface) protocol, the module is started as an independent thread. This allows the server to handle multiple clients, which will be useful when multiple workloads are connected to the simulation. However, implementing the module into the system was challenging, so we decided to start it independently. In earlier versions, we attempted to implement it inside the ecovisor model, but doing so halted the execution of the simulation until the API-Server was stopped, rendering the system unusable. Starting the API-Server independently also allows it to be scaled independently from the ecovisor model and the RedisDB. This may be helpful in bigger simulations with distributed infrastructure. We did not implement the *tick()* function from the ecovisor API since it is not used in our approach, and Mosaic already includes a time mechanism for the simulation part. Therefore, we have decided not to implement the `tick()` endpoint to the API for now. However, we can easily add this functionality at a later time.

2) RedisDB: The RedisDB is a high-speed, in-memory, key-value data store that is used in our system to store and retrieve simple data types within the ecovisor model and provide them to the API-Server. To simplify simulation operation, we start the RedisDB as a Docker Container using the docker python library [25], which integrates the docker engine API into python. We use the Redislabs Rejson Redis image [26], which expands the RedisDB image's functionality to handle JSON values. The Redis container is started at the beginning of the simulation to ensure it is operational when the first data is available. When the simulation ends, we stop and delete the container to keep the test environment clean. The Ecovisor-Redis-Interface in the ecovisor model can be adapted to replace the database with any other database, which may be useful for integrating the simulation into a production-grade cloud environment like Kubernetes or Openstack.

3) Ecovisor-Redis-Interface: The interface to RedisDB is implemented within the ecovisor model using the Redis-py library [27] to simplify access. The ecovisor model has two methods: `get_redis_updates()` and `send_redis_update()` which are used to interact with RedisDB.

The `get_redis_update()` method is called at the beginning of the `step()` method to update the simulation. This method is responsible for updating the values of `battery_discharge_rate`, `battery_charge_level`, and `container_power_caps`. These updated values are then used in subsequent power calculations within the ecovisor model. To facilitate the exchange of data with the RedisDB, the data structure presented in Algorithm 2 is used. All the values in this data structure are retrieved from the RedisDB, but only the values mentioned above are updated and utilized in the simulation. Moreover, the data structure 3 is employed to update the `container_power_cap` of various workload applications. This process helps to maintain the desired level of power consumption in the system and keep it within the specified limits.

```
{
    "solar_power": "0 kW",
    "grid_power": "0 kW",
    "grid_carbon": "0 g · CO2/kW",
    "battery_discharge_rate": "0 kW",
    "battery_charge_level": "0 kWh"
}
```

Algorithm 2: Energy Data JSON

```
{
    "Container_ID 1": "Power Cap 0 kW",
    "Container_ID 2": "Power Cap 0 kW",
    "Container_ID n": "Power Cap ..."
}
```

Algorithm 3: Container Power Cap JSON

The `send_redis_updates()` method is called at the end of the `step()` method, which publishes the updated values from the ecovisor model. This includes data from other parts of the simulation that are accessible by the ecovisor model. The data is exchanged with the RedisDB using the data structure shown in Algorithm 2. Once the data is in RedisDB, it can be accessed by the workload application via the API-Server.

4) Dataflow: In Figure 3, we can see the data exchange between the API-Server and the ecovisor model, with two

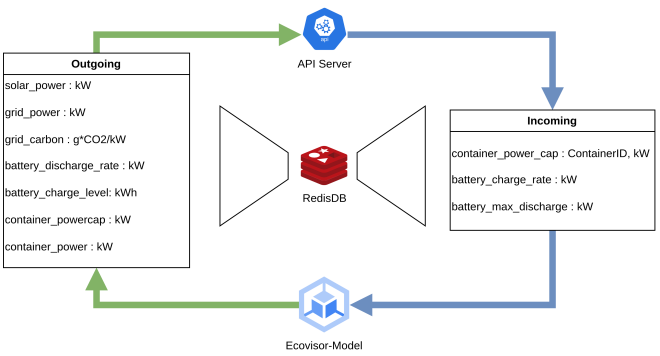

Figure 3. Dataflow between ecovisor model and API Server

separate data streams: upstream from the API-Server to the ecovisor model, and downstream from the ecovisor model to the API-Server. The values written in each endpoint are not duplicated in the other endpoint. The workload applications access the setter methods from Table I, which only affect the values `container_powercap`, `battery_charge_rate`, and `battery_max_discharge`. These values are only read in the ecovisor model. The values written by the ecovisor model are only accessible via the getter-methods of the API. This setup ensures data consistency, since none of the values can be written in parallel. Although the RedisDB theoretically supports atomic writes to ensure data-consistency [23], we do not require this feature in our setup.

To model the entire data flow, the ecovisor model receives updates from the RedisDB, processes the data, and sends the updated data back to the RedisDB so that it can be accessed by the workload application. Any updates sent by the workload application will be processed no later than the next call of the `step()` method. This process encompasses the complete data flow between the ecovisor model, the API server, and the workload applications.

V. Evaluation

To assess the effectiveness of our system, we conducted a two-part evaluation using open-source real-world data. We designed two example cases to demonstrate the simulation capabilities of Ecovisor and showcase how the implemented API can be used by consumers. To improve the clarity of the data presented in our examples, we scaled down the Photo Voltaic (PV) data to approximately 30% of its original values. This allowed us to better showcase the API usage and plotted graphs. Scaling down the input data did not impact the simulation itself, as we could simply assume that less solar power was available due to weather or limitations of the solar power generators.

In the first part of our evaluation, we demonstrate the general functionality of the Ecovisor model and showcased how the output changed with varying input of the energy mix. In the second example, we highlighted a missing functionality of the API by demonstrating how a workload application can set the `battery_charge_rate` and `battery_max_discharge`.

To enhance the comprehensibility of our evaluation, we included two plots visualizing the output data from the simulation. One plot was included for each example, helping to clearly illustrate the results of our evaluation.

A. Ecovisor-Model with changing energy input

The first example showcased the general functionality of the Ecovisor model and how the output is affected by changing availability of different energy sources. For this simulation, we utilized an example carbon data to determine the amount of carbon emitted when using grid energy and a solar energy input file, which was scaled down to one-third of its original value. Starting from second 200, we further scaled it down until the output was under the total consumption of the workload application. In addition, we set the `battery capacity` to 0.3 KW/h and the `battery_charge_level` to 0.15 KW/h.

In the plot 4, we can observe that the `Grid Power` output is opposite to the PV Power output until second 200. This is because when enough PV energy is available, the surplus energy is fed into the energy grid, resulting in negative carbon emission. During the first 200 seconds, the workload application's consumption is stable at 500 KWs and doesn't change throughout the simulation. Although this isn't ideal in a real-world scenario, it allowed us to more easily demonstrate the functionality of the model.

In the first third of the simulation, until second 100, we can see that there is enough PV energy to run the workload and feed some extra energy into the grid. However, the Battery Charge Level isn't high enough to load the battery consistently. Around second 100, the PV Energy level rises, and the battery starts charging until second 200. At second 200, the PV Energy level drops, and the workload application uses the energy in the battery and from the available PV energy to meet its demands. Around second 350, no PV Energy is available, and the workload only runs on grid power until the end of the simulation. The battery doesn't charge since it only charges with PV energy.

This example demonstrates how the Ecovisor model behaves with different available energy sources. It charges the battery when enough PV energy is available and uses all other available energy before utilizing grid energy to reduce carbon emission.

B. Additional API Functionality

The second showcase, illustrated in Figure 5, employs the same PV and Carbon data as in the first showcase, resulting in the same correlation between PV Energy and Grid Energy. The objective of this showcase is to demonstrate the usage of the API endpoints for setting the `container_powercap`, the `battery_charge_rate`, and the `battery_max_discharge`.

While the `container_powercap` was also utilized in the first showcase, the consumption set in the beginning remained constant. In this showcase, we developed a Python script that emulates a workload application by sending API requests to adjust the `container_powercap` around second 45 from 500 KWs to 250 KWs, which is clearly discernible in the plot. This capability enables a workload application to dynamically adjust its power consumption while scaling up or down.

Furthermore, the `battery_charge_rate` and `battery_max_discharge` are utilized to operate the battery. The `battery_charge_rate` determines how much of the available PV Energy is used to charge the battery, while the `battery_max_discharge` sets the threshold of energy that will not be used. These parameters can be adjusted to either extend the battery's lifespan or to implement different energy usage profiles. We changed both values by sending API requests with the same Python script used to set the `container_powercap`. At around second 75, we can observe that after setting both battery control

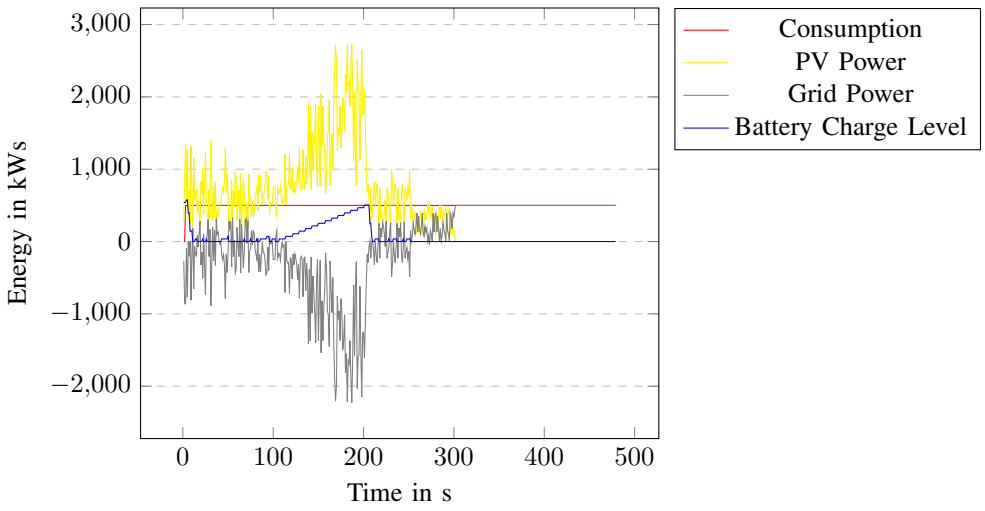

Figure 4. Showcase changing Energy Mix

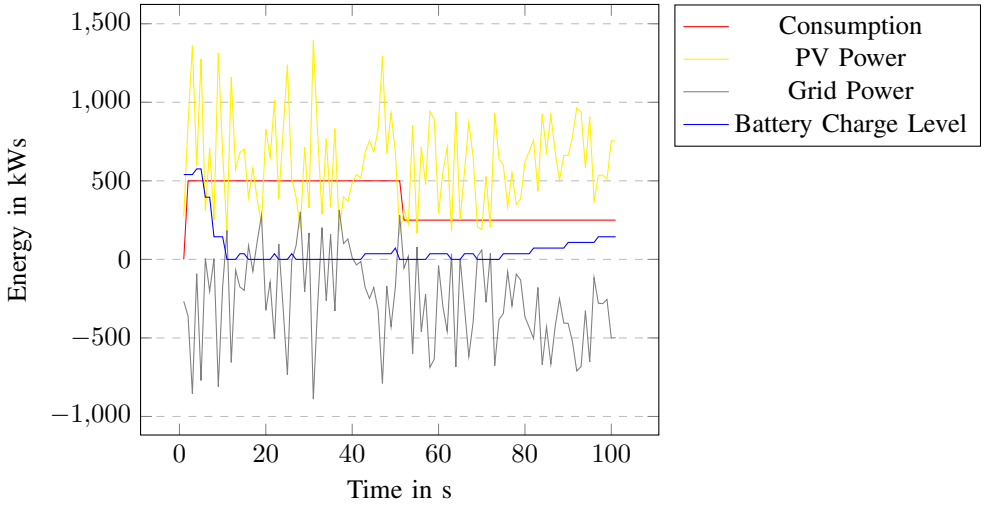

Figure 5. Showcase additional API Functionality

values to 10 KWs, the battery begins to charge even when the available energy does not change.

These two examples demonstrate the functionality of the Ecovisor-Model and its API. They show how the model can be used to simulate an Ecovisor for a workload application, allowing researchers and engineers to test different energy usage profiles and settings.

VI. Future Work

While our approach focused on a single simulation of the ecovisor, our use of Mosaik demonstrates its effectiveness for large-scale smart grid simulations. In future research, we suggest interconnecting multiple ecovisor systems to share resources and further reduce carbon emissions. This network could be distributed across different geographic regions, as carbon intensity varies depending on location. By incorporating carbon information services like Electricity Maps [28], this could enable carbon-efficiency optimizations such as Let's Wait Awhile [29] or Cucumber [30] from Wiesner et al. This would enhance the potential for carbon reduction at a larger scale. This work also does not focus on monitoring and limiting the power consumption of applications in the loop, due to the specific requirements of the implementation. Therefore, a potential future direction could involve incorporating an interface that enables various power measurement and control techniques for different application hosts, such as containers, virtual machines, or physical hardware systems.

VII. Conclusion

In this article, we simulated Souza et al.'s [10] ecovisor and integrated it into a Mosaik co-simulation. This design abstracts the physical implementation and allows for the interaction with various simulators, such as simulated virtual energy resources or simulated consumers. To integrate real applications into this environment, we have enabled SIL and HIL capabilities by exposing the ecovisor API through a FastAPI server that is connected to a Redis database. We have devised a cost-effective and time-efficient approach for testing and developing carbon-aware applications that does not require a physical ecovisor implementation. In order to assess our approach, we

generated test cases by utilizing recorded solar and carbon data. These tests aimed to showcase the ecovisor model's implementation accuracy and its capability to transfer data correctly between the model and other systems.

REFERENCES

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[2] G. Brockman, "Chatgpt just crossed 1 million users," Twitter, dec 2022, accessed: 16.02.2023. [Online]. Available: https://twitter.com/gdb/status/1599683104142430208?s=20&t=-7V8elwpZ93qiIhwxnzVBA

[3] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of internet services and applications*, vol. 1, pp. 7–18, 2010.

[4] M. A. B. Siddik, A. Shehabi, and L. Marston, "The environmental footprint of data centers in the united states," *Environmental Research Letters*, vol. 16, no. 6, p. 064017, 2021.

[5] Gartner Inc., "Gartner forecasts worldwide public cloud end-user spending to reach nearly $600 billion in 2023," oct 2022, accessed: 16.02.2023. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2022-10-31-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-600-billion-in-2023

[6] Google LLC, "Building a carbon-free future for all," accessed: 16.02.2023. [Online]. Available: https://sustainability.google/commitments/carbon/

[7] Amazon.com, Inc., "Sustainability in the cloud," accessed: 16.02.2023. [Online]. Available: https://sustainability.aboutamazon.com/environment/the-cloud

[8] N. Scarlat, M. Prussi, and M. Padella, "Quantification of the carbon intensity of electricity produced and used in europe," *Applied Energy*, vol. 305, p. 117901, 2022.

[9] S. Murugesan, "Harnessing green it: Principles and practices," *IT Professional*, vol. 10, no. 1, pp. 24–33, 2008.

[10] A. Souza, N. Bashir, J. Murillo, W. Hanafy, Q. Liang, D. Irwin, and P. Shenoy, "Ecovisor: A virtual energy system for carbon-efficient applications," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 252–265.

[11] C. Steinbrink, M. Blank-Babazadeh, A. El-Ama, S. Holly, B. Lüers, M. Nebel-Wenner, R. P. Ramírez Acosta, T. Raub, J. S. Schwarz, S. Stark, A. Nieße, and S. Lehnhoff, "Cpes testing with mosaik: Co-simulation planning, execution and analysis," *Applied Sciences*, vol. 9, no. 5, 2019.

[12] M. Vogt, F. Marten, and M. Braun, "A survey and statistical analysis of smart grid co-simulations," *Applied Energy*, vol. 222, pp. 67–78, 2018.

[13] T. Kelemenová, M. Kelemen, L. Miková, V. Maxim, E. Prada, T. Lipták, and F. Menda, "Model based design and hil simulations," *American Journal of Mechanical Engineering*, vol. 1, no. 7, pp. 276–281, 2013.

[14] OFFIS – Institut für Informatik, "Mosaik 3.1.1 documentation," accessed: 16.02.2023. [Online]. Available: https://mosaik.readthedocs.io

[15] M. H. Cintuglu, O. A. Mohammed, K. Akkaya, and A. S. Uluagac, "A survey on smart grid cyber-physical system testbeds," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 1, pp. 446–464, 2017.

[16] J. Mambretti, J. Chen, and F. Yeh, "Next generation clouds, the chameleon cloud testbed, and software defined networking (sdn)," in *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*, 2015, pp. 73–79.

[17] N. Mansouri, R. Ghafari, and B. M. H. Zade, "Cloud computing simulators: A comprehensive review," *Simulation Modelling Practice and Theory*, vol. 104, p. 102144, 2020.

[18] J. Beilharz, P. Wiesner, A. Boockmeyer, F. Brokhausen, I. Behnke, R. Schmid, L. Pirl, and L. Thamsen, "Towards a staging environment for the internet of things," in *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 2021, pp. 312–315.

[19] V. Hagenmeyer, H. Kemal Çakmak, C. Düpmeier, T. Faulwasser, J. Isele, H. B. Keller, P. Kohlhepp, U. Kühnapfel, U. Stucky, S. Waczowicz, and R. Mikut, "Information and communication technology in energy lab 2.0: Smart energies system simulation and control center with an open-street-map-based power flow simulation example," *Energy Technology*, vol. 4, no. 1, pp. 145–162, 2016.

[20] OFFIS – Institut für Informatik, "mosaik-components contains some components, which can be used as base for a co-simulation scenario," accessed: 16.02.2023. [Online]. Available: https://gitlab.com/mosaik/components

[21] A. Bourdon, A. Noureddine, R. Rouvoy, and L. Seinturier, "Powerapi: A software library to monitor the energy consumed at the process-level," *ERCIM News*, vol. 2013, no. 92, 2013.

[22] S. Ramírez, "FastAPI framework, high performance, easy to learn, fast to code, ready for production," accessed: 03.03.2023. [Online]. Available: https://fastapi.tiangolo.com

[23] Redis Ltd, "Redis – a vibrant, open source database," accessed: 03.03.2023. [Online]. Available: https://redis.io

[24] Encode UK Ltd, "Uvicorn – an ASGI web server, for python," accessed: 03.03.2023. [Online]. Available: https://www.uvicorn.org

[25] Docker Inc, "Docker SDK for python," accessed: 03.03.2023. [Online]. Available: https://github.com/docker/docker-py

[26] Redis Ltd, "RedisJSON – enhanced json data type processing for redis," accessed: 03.03.2023. [Online]. Available: https://hub.docker.com/r/redislabs/rejson

[27] ——, "redis-py – the python interface to the redis key-value store," accessed: 03.03.2023. [Online]. Available: https://github.com/redis/redis-py

[28] Electricity Maps A/S , "Reduce carbon emissions with actionable electricity data," accessed: 16.02.2023. [Online]. Available: https://www.electricitymaps.com/

[29] P. Wiesner, I. Behnke, D. Scheinert, K. Gontarska, and L. Thamsen, "Let's wait awhile," in *Proceedings of the 22nd International Middleware Conference*. ACM, dec 2021.

[30] P. Wiesner, D. Scheinert, T. Wittkopp, L. Thamsen, and O. Kao, "Cucumber: Renewable-aware admission control for delay-tolerant cloud and edge workloads," in *Euro-Par 2022: Parallel Processing*. Springer International Publishing, 2022, pp. 218–232.