








RESEARCH ARTICLE

Open Energy Services - Forecasting and Optimization as a Service for Energy Management Applications at Scale

David Wölflé¹, Kevin Förderer², Tobias Riedel¹, Natascha Fernengel², Lukas Landwich¹, Ralf Mikut², Veit Hagenmeyer² and Hartmut Schmeck¹

¹Intelligent Systems and Production Engineering, FZI Research Center for Information Technology, Haid-und-Neu-Str. 10–14, 76131 Karlsruhe, Germany

²Institute for Automation and Applied Informatics, Karlsruhe Institute of Technology, Hermann-von-Helmholtz-Platz 1, 76344 Eggenstein-Leopoldshafen, Germany

*Corresponding author. E-mail: woelfle@fzi.de

Received: 05 August 2024

Keywords: software framework; building energy management; building energy optimization; building control; model predictive control; forecasting; smart building

Abstract

This article aims at facilitating the widespread application of **Energy Management Systems (EMSs)**, especially on buildings and cities, in order to support the realization of future carbon-neutral energy systems. We claim that economic viability is a severe issue for the utilization of EMSs at scale and that the provisioning of forecasting and optimization algorithms as a service can make a major contribution to achieve it. To this end, we present the *Energy Service Generics* software framework that allows the derivation of fully functional services from existing forecasting or optimization code with ease. This work documents the strictly systematic development of the framework, beginning with a requirement analysis, from which a sophisticated design concept is derived, followed by a description of the implementation of the framework. Furthermore, we present the concept of the *Open Energy Service* community, our effort to continuously maintain the service framework but also provide ready-to-use forecasting and optimization services. Finally, an evaluation of our framework and community concept, as well as a demarcation between our work and the current state of the art, is presented.

Impact Statement

Energy management will likely play a vital role in future carbon-neutral energy systems, as it allows for unlocking energy efficiency and flexibility potentials. However, energy management systems need to be applied at large scales to realize the desired effect, which clearly requires minimization of costs for setup and operation. We promote an approach to split the complex optimization algorithms employed by energy management systems into standardized components, which can be provided as a service with marginal costs at scale. This work introduces a framework as well as a community concept to support the efficient implementation and operation of such services. Thus, this work is a significant step towards the large-scale application of energy management systems aiding a carbon-neutral future.

1. Introduction

Global scale efforts are required to mitigate the most severe consequences of climate change, including a significant increase in the energy efficiency of consumers as well as the decarbonization of energy supply [IPCC, 2022]. The vast utilization of renewable energy sources required for the latter will additionally likely induce an increased demand for energy flexibility by consumers [Alizadeh et al., 2016, Kondziella and Bruckner, 2016, Papaefthymiou and Dragoon, 2016]. EMSs, in a sense of software computing optimized operational schedules and executing these on devices and systems, have been demonstrated to be capable of reducing energy demand, lowering CO_2 emissions and/or unlocking flexibility [Schibuola et al., 2015, Oldewurtel et al., 2012, Ding et al., 2019, Salpakari and Lund, 2016, Chen et al., 2019]. However, in order to achieve the desperately needed global impact energy management solutions will be required at scale, like e.g. applied to thousands of buildings.

Economic viability is certainly a key factor for the widespread adoption of EMSs. Forecasting and optimization algorithms are essential parts of EMSs (see Section 2), but have traditionally been developed for a single specific target (e.g. for one particular building) [Wölflle et al., 2020], like in [Schibuola et al., 2015, Oldewurtel et al., 2012, Ding et al., 2019, Salpakari and Lund, 2016, Chen et al., 2019] or the publications reviewed by [Shaikh et al., 2014]. This approach is problematic as it has been shown that the development costs of target-specific forecasting and optimization algorithms are higher than the monetary savings, even for medium-sized commercial buildings [Gwerder et al., 2013].

This paper aims at supporting the widespread adoption of EMSs by enabling the utilization of forecasting and optimization algorithms for energy management applications at large scales. Our approach, as discussed in Section 2 in detail, is to replace target-specific forecasting and optimization algorithms (which are locally deployed as part of the EMS instances) with generic forecasting and optimization algorithms that are centrally provided as web services, in order to reduce development and operation costs of EMSs. In Section 3 we extensively analyze the current state of the art and find that the concept of providing forecasting and optimization algorithms as web services is already well established, especially in commercial solutions provided by international corporations. Furthermore, it is relevant to note that data-driven algorithms, i.e. forecasting and optimization approaches generally suitable for utilization in larger scales of EMSs controlling heterogeneous systems, have been frequently proposed in academia [Anand et al., 2023, Chen et al., 2019, Ding et al., 2019, Meisenbacher et al., 2023, Xuereb Conti et al., 2023]. However, it seems that currently no software framework exists that supports the implementation and operation of such services, which seems to be a major barrier for bringing these new and innovative forecasting and optimization algorithms into practical application by EMSs.

The present paper addresses the aforementioned shortcoming by contributing a *framework* that allows the provisioning of forecasting or optimization code as a web service. To that end, we begin by carrying out an extensive analysis to specify requirements (Section 4). Based on this, we present a sophisticated design concept that satisfies these requirements (Section 5) and finally derive an implementation of our concept (Section 6), which we release as a free and open source repository alongside this publication. Our second contribution is the presentation of our concept for the *Open Energy Services* community (Section 7), a group that is dedicated to the maintenance of the framework, but also to the development and operation of forecasting and optimization services. Finally, Section 8 is devoted to demonstrating that our contributions, i.e. framework and community concept, are useful for facilitating the development and operation of forecasting and optimization services for energy management applications.

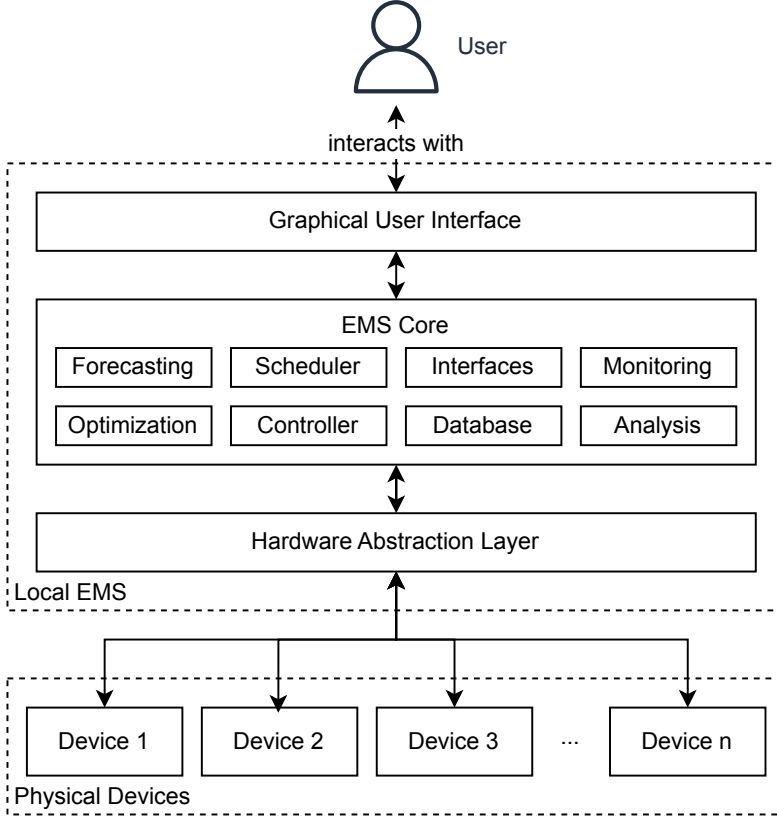


Figure 1. Typical high-level architecture of an *EMS*.

2. Nomenclature

As a first step to define the context this work is set in, we begin with inspecting the typical *Energy Management System (EMS)* architecture¹. Concrete proposals for the latter have been provided by [Dawson-Haggerty et al., 2013, Lee et al., 2016, Mauser et al., 2015, Pipattanasomporn et al., 2015]. [Han et al., 2023] contains a review of architectures of *EMS* for residential buildings. While these papers generally show no consensus about the internal structures of *EMS*, it is nevertheless easily possible to map the respective suggested architectures to the convention introduced below and summarized in Figure 1. The latter also holds for the architecture of OpenEMS², the only *EMS* the authors are aware of that is developed as an open source project by a consortium of commercial institutions.

In order to discuss the internal structures of *EMSs* in greater detail, we consider a running example of a commercial building equipped with a *Photovoltaic (PV)* and *Battery Storage System (BSS)* as *physical devices* as well as an *EMS*, intended to optimize the operation of the latter, that is executed on a computing device inside the building. As the facility managers of the building are responsible for its correct operation, they interact with the *EMS*, e.g. to monitor the operation or to adjust setpoints. However, other residents of the building may

¹It is worth noting that many commercial offerings of *EMSs* are regularly reduced to an implementation of ISO 50001, which defines energy management applications with a focus on monitoring and analysis. Such *EMSs* will usually not contain forecasting and optimization components. Thus, they are not referenced here as they are not covered by the scope of this work. However, the service approach promoted in this work might in fact be a relevant option for unlocking to potential of forecasting and optimization algorithms for such *EMSs*.

²<https://openems.github.io/openems.io/openems/latest/edge/architecture.html>

interact with the EMS too, e.g. to specify their personal demands which the system should consider. Thus, all persons interacting with the EMS are the *users* of it. The EMS has been developed and is supported by a specialized institution, the *EMS developer*.

The EMS itself is essentially a piece of software consisting of three major parts:

1. A *Graphical User Interface (GUI)* which the users interact with.
2. A *Hardware Abstraction Layer (HAL)* connecting the EMS to the physical devices.
3. A component holding the essential management functionality, which we will refer to in this work as *EMS core*.

Returning to our running example, we can perceive the functionality of the EMS core part to contain:

- *Optimization*: Computes optimized *schedules* for the controllable devices in order to satisfy the goals provided by the users. E.g. the facility manager could configure the EMS such that the BSS is used to take advantage of flexible electricity tariffs.
- *Forecasting*: Computes *forecasts* that the optimization algorithm requires as input. In the present example, the optimization could require predictions of the future development of the energy price, the electric load, and the power generation of the PV system.
- *Scheduler*: Invokes the forecasting and optimization algorithms periodically or at certain events. In the present example, the scheduler might trigger the computation of an optimized schedule for the BSS every 15 minutes by first invoking the forecasting algorithms and then forwarding the predictions (along with any other required input data) to the optimization algorithm. The scheduler might additionally fetch data from external sources, like e.g. a weather forecast as necessary input for a PV power prediction algorithm.
- *Controller*: Ensures that the user and hardware constraints are satisfied by the EMS. For example, the facility manager could wish to enforce that the BSS is not discharged below 20% in order to expand the lifetime of the device. The controller might additionally contain simple rules that define a sane default strategy in case that the optimization algorithm does not work as intended.
- *Database*: Stores the data required for the operation of the EMS (incl. GUI), like, for example, measurements emitted by the physical devices.
- *Interfaces*: Implements the connectivity to the GUI and the HAL. Might additionally contain interfaces for external applications or a message broker for internal communication between parts of EMS core.
- *Monitoring*: Continuously oversees the system influenced by the EMS and emits alerts in case of malfunctioning. The monitoring system could, for example, send an email to the facility manager if the communication with devices has been lost or these need maintenance.
- *Analysis*: Aggregates and computes metrics relevant for the users of the EMS, for example statistics about the energy usage pattern.

In contrast to the usual EMS architecture pattern introduced above, this work promotes an approach in which the forecasting and optimization algorithms are not directly integrated into the EMS, but provided as services, as summarized in Figure 2. It is worth noting that, although forecasting and/or optimization algorithms are utilized as a service, the correct operation of the EMS remains the responsibility of the EMS developer, i.e. by implementing a controller component (see above) into the EMS.

In the context of this work, a *service* refers to a web-based program that provides a functionality required for energy management applications via a standardized interface. The intention of providing forecasting and optimization algorithms as services is to make these algorithms available to a larger number of EMSs in order to reduce development and maintenance costs of the individual systems. The separation of forecasting and optimization algorithms from the

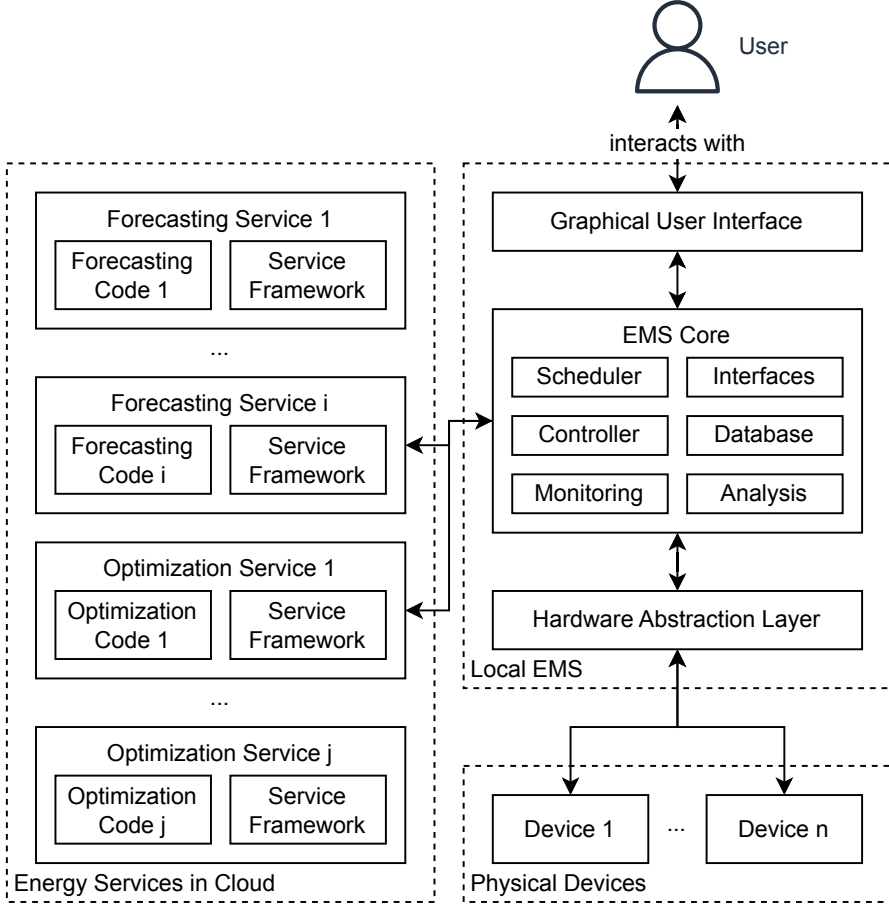


Figure 2. High-level architecture of an EMS utilizing selected forecasting and optimization services.

EMS software implies the need to extend the former with interfaces in order to allow the interaction between the services and the EMSs. Furthermore, the intended usage of the forecasting and optimization algorithms by a large number of EMSs makes it necessary to consider how these can be executed in a scalable way. For example, one should consider that the implementation of a forecasting or optimization algorithm, henceforth referred to as the *forecasting or optimization code*, will generally not contain an **Application Programming Interface (API)** suitable for web-based clients or functionality to concurrently handle thousands of requests. Thus, it is necessary to extend the forecasting or optimization code, with all the functionality required for an operation as a service. However, it is obviously not very effective to develop and implement this extension for every service from scratch, as it will likely be very similar for all services. Hence, the utilization of a *service framework*, i.e. a software that drastically reduces the necessary effort for developing forecasting and optimization services, by providing the software parts that are generic for all services. In fact, a large fraction of this work is devoted to the design and implementation of such a service framework.

It should be noted that this service framework is by no means limited to services for forecasting and optimization: Consider e.g. a heuristic that detects the occupancy in a building from limited information, or an algorithm (like in [De Jongh et al., 2022]) that determines the current state of the electricity grid. The provisioning of such algorithms as services is clearly

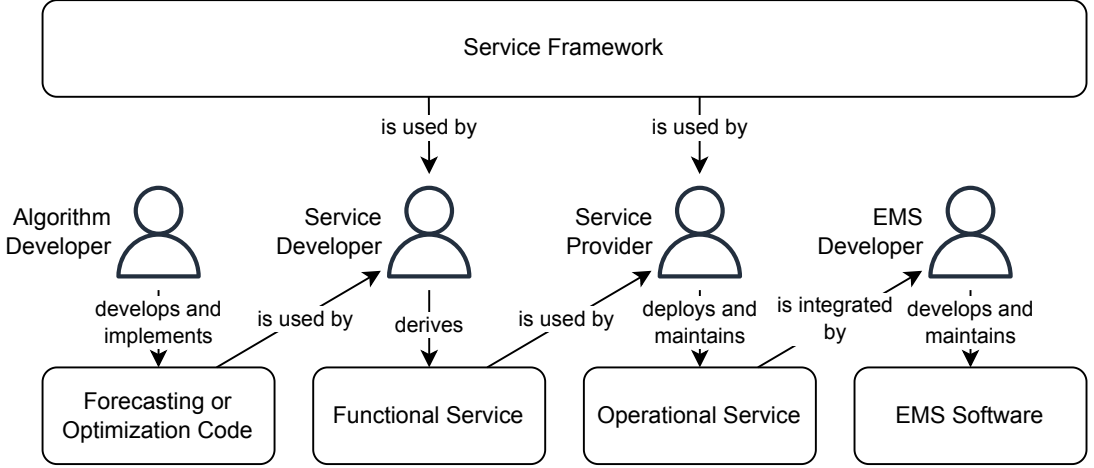


Figure 3. Stakeholders involved in the development process of forecasting or optimization services.

useful in the wider sense of energy management. Furthermore, by its generic design, the proposed framework is applicable for use cases not related to energy management. Consider, for example, the flood prediction approach proposed by [Hofmeister et al., 2024], which could be provided as a service, too. However, in the following, for simplicity and readability, we refer to *forecasting and optimization* or to the retrieval of a *forecast or optimized schedule*. This is not meant to exclude other, not strictly covered, but related algorithms.

Finally, it is necessary to regard the development process of a forecasting and optimization service as well as the corresponding stakeholders that are involved. The first step is the development and implementation of the forecasting or optimization algorithm by the *algorithm developer*. This step might have been finished far before the development of a service has been decided and we thus use the terminology of an *existing forecasting or optimization code*, in order to illustrate that no considerations about a potential utilization of the code in a service need to be taken during the development. The subsequent step of the development process is carried out by the *service developer*, who wraps the existing forecasting or optimization code with the service framework in order to *derive* a *functional service*. The latter is then *operated* by the *service provider* to make it usable for EMSs. It is worth noting that the service framework contains an operation concept (see Section 5.4) which supports service providers with their tasks. The integration of the service into the EMS is carried out by the *EMS developer* who additionally needs to negotiate with the service provider with respect to the conditions under which the service can be used, including which data the EMS must provide to the service. The job of installation and maintenance of the EMS is carried out by the *EMS provider*, likely in close cooperation with the final *user* of the system.

Returning to our running example, one might consider that the algorithm developer is an academic researcher who engineered the optimization algorithm for the BSS within a project funded by an IT company that specializes in selling forecasting and optimization services. The latter might act in the roles of the service developer and service provider, having several customers that specialized in designing and provisioning of EMS. The facility manager might then have ordered an EMS for the building they supervise and therefore become the user of the EMS, thus also indirectly the user of the integrated services.

Finally, it is worth mentioning that we do consider, but not demand, that the aforementioned roles are distributed over institutions. Nevertheless, it appears not unlikely in academic research

projects that all roles are taken by a single institution, e.g. a research group, who might develop algorithms as well as an EMS and test it in their own research facilities.

3. Related Work

This section analyses approaches related to ours from academia and industry.

3.1. Frameworks for Service Development

Most closely related to the present publication is [Maree and Bagle, 2022], in which a service-based approach to create digital twins of buildings is presented. Similar to our work, the paper strives to develop a framework. The main difference is that their work is much broader, i.e. the framework covers not only forecasting and optimization but also data storage, thermal models of buildings, and how these can be learned from data. Consequently, [Maree and Bagle, 2022] do not handle the aspect of forecasting and optimization services at a comparable depth as in the present work. For example, it does not contain any requirements analysis, a detailed discussion about the technical design and implementation of the services, nor does it provide a systematic approach to derive new forecasting or optimization services from existing code. Furthermore, their work does not contain any hint about a potential publication of the corresponding source code. Thus we conclude that their work, unlike ours, is not a reasonable basis for deriving forecasting and optimization services for energy management applications.

On the other hand, larger Machine Learning (ML) frameworks, like e.g. PyTorch³ MLflow⁴, provide the functionality to expose an ML model as web service with a Representational State Transfer (REST) API. However, to the best of our knowledge, there is no solution that supports triggering the training models from API calls out of the box⁵, i.e. that allows fitting system-specific parameters as our framework does. It thus appears that utilizing the framework developed in the present work is significantly more advantageous for service developers. This is particularly the case if one considers that our framework has been explicitly designed to minimize the necessary effort for the development and operation of forecasting and optimization services for EMSs at scale.

3.2. Forecasting and Optimization Services

Several publications have been identified, beyond [Maree and Bagle, 2022], that utilize the concept of forecasting and/or optimization components wrapped into services [Galenzowski et al., 2023, Lenk et al., 2020, Hill et al., 2023, Dengler et al., 2023, Marinakis et al., 2020, Mohamed et al., 2018]. Regarding approaches not documented in academic publications one should first consider that several providers exist operating web APIs for the retrieval of weather-related data or forecasts, partly as free or commercial offering, e.g. Bright Sky⁶, Open Meteo⁷, SoDa⁸, Solcast⁹ and Forecast.Solar¹⁰. The latter two offer additional services related to forecasting PV power generation. Closely related to the latter is NIXTLAs TimeGPT¹¹, a commercial service for generic time series forecasting. Finally, it is

³<https://pytorch.org/>

⁴<https://mlflow.org/docs/latest/models.html>

⁵A detailed discussion why training models via API calls is necessary is provided in Section 4.2.

⁶<https://brightsky.dev/>

⁷<https://open-meteo.com/>

⁸<https://www.soda-pro.com/>

⁹<https://solcast.com/>

¹⁰<https://forecast.solar/>

¹¹https://docs.nixtla.io/docs/getting-started-about_timegpt

worth mentioning the Building Energy Modeling¹² service provided by Schneider Electric as part of their EcoStruxure platform, more details about the latter in the following section. The service allows users to learn the thermal energy consumption pattern of buildings from data.

In contrast to our work, none of the publications or services referenced in this section present a framework for deriving forecasting and optimization services for energy management applications. Thus, these services are neither in conflict with the present work nor do they provide any substantial input for the requirements analysis or design concept presented below. While some of the mentioned offerings could be reasonably utilized by EMSs, the main issue is that each of these services covers only a fraction of the typically required functionality, while none provides a generic approach to derive the remaining necessary forecasting and optimization services. Nevertheless, the pure existence of these publications and services can be considered as strong advocacy for the general concept of forecasting and optimization services and, thus, the relevance of the present paper. Furthermore, it is worth noting that the scientific research underlying the referenced publications would very likely have benefited substantially from using the framework proposed in the present work.

3.3. Energy Management Systems, Platforms, Communities and Market Places

A relevant platform that has been developed and utilized for years in EU projects is FIWARE [Cirillo et al., 2019]. FIWARE is a general-purpose IoT Platform [Cirillo et al., 2019] managed by the FIWARE Foundation [Rodriguez et al., 2018]. It is used in various use cases, including smart farming [Rodriguez et al., 2018], smart buildings, and smart grids [Blechmann et al., 2023]. The heart of FIWARE is the so-called context broker, which receives data from data providers (e.g., sensors), stores the latest information, and provides it to data consumers (e.g., some service). Aside from the context broker, there are various different solutions for data processing and storage that can be connected to the broker, as well as a set of “smart data models” that have been used in different applications. With all these, the FIWARE ecosystem provides many different building blocks that can be used in energy management. These building blocks, however, are very heterogeneous and generic, as their only shared foundation is the integration with the context broker and the underlying Next Generation Service Interface (NGSI). Therefore, FIWARE is less an alternative to the proposed service framework and more a platform into which derived services could be integrated.

On the other hand, several commercial approaches exist that are similar to our service-based forecasting and optimization concept. In particular noteworthy are the platform solutions from Siemens (Building X¹³), Bosch (NEXOSPACE¹⁴) and Schneider Electric (EcoStruxure¹⁵). The latter two appear conceptually similar, i.e. the platforms provide several functionalities for smart building operation, including energy management, but require that proprietary hardware (i.e. gateways) must be installed in the building that should be connected to the respective platform. This is a clear contrast to the Siemens solution, which is advertised with an open API concept and connectivity to third-party systems, while seemingly offering similar functionality like the other two. No evidence was found that any of the three companies offers a framework like introduced in this work. However, all three vendors claim that their platform solutions can be extended by third parties and offer a marketplace for applications that can be integrated. However, publishing extensions on the marked places must be explicitly granted by the respective company and is subject to licensing fees. Thus, we conclude that none of the three platforms is indeed a viable alternative to this work as neither empowers third parties to develop and operate forecasting and optimization services for energy management applications

¹²<https://exchange.se.com/develop/products/38969/building-energy-modeling-api>

¹³<https://xcelerator.siemens.com/global/en/products/buildings/building-x.html>

¹⁴<https://www.boschbuildingsolutions.com/x/en/digital-services/>

¹⁵<https://www.se.com/ww/en/work/campaign/innovation/platform.jsp>

which are independent of the respective vendor. Finally, it is worth mentioning that Schneider Electric and Bosch both offer EMSs for private households, e.g. Bosch Smart Home¹⁶ and HEM-Slogic¹⁷. While both of these systems apparently use some form of cloud-based optimization, there seems to be no possibility to directly interact with these forecasting and optimization services or to integrate third party services as an alternative. However, we again perceive that the existence of the Siemens, Schneider Electric and Bosch solutions, providing cloud services for smart building operation and energy management, strongly advocates the concept underlying this work.

Finally, we find it important to discriminate our work from the Open Energy Platform¹⁸ as well as from OpenEMS¹⁹, two projects well known among scientific researchers. The first of these is a community effort to establish a collection of tools supporting the work with and publication of energy-related datasets, with a focus on energy system modeling. This is clearly disjoint from our goal to provide tooling for the implementation of forecasting and optimization services for energy management applications. On the other hand, OpenEMS is a fully functional, open source, and free to use EMS. While it does, in fact, contain a limited number of forecasting and optimization algorithms, providing these is not the essential task of the software. The latter is particularly true as OpenEMS is usually operated on edge devices with little compute power, which limits the applicability of modern ML-based forecasting and optimization methods. However, it is absolutely reasonable to extend OpenEMS with an Energy Service Generics (ESG) compatible client, to allow the integration of forecasting and optimization services derived with our framework, and we plan to demonstrate this in future work.

4. Requirements Analysis

Following the common procedure in software engineering, we begin with a systematic approach to assess the requirements that should be fulfilled by our service framework. IEEE defines a requirement as 'A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents' [IEEE, 2002]. It is worth noting that the traditional requirements engineering process, as defined in [Pohl, 1996], is tailored for the utilization in customer-specific software development. In contrast, this work aims at developing a framework for a broad range of potential service developers, from academia and industry alike. We thus employ a simple two-step process inspired from research on market-driven requirements engineering [Alves et al., 2006, Regnell and Brinkkemper, 2005]. Hereby, the first step is an analysis of application areas. To this end, typical applications of EMSs in three different areas are described in Section 4.1. Building upon this information, the requirements are documented in a semi-structured natural language specification [Washizaki, 2024], using the following pattern: 'An <actor> must/should be able to <requirement>'. Using this pattern, the requirements are documented in a transparent and consistent way that enables easy requirements verification. Furthermore, we categorize the requirements into functional (Section 4.2) and non-functional (Section 4.3). Here we follow Glinz's definition [Glinz, 2007], where functional requirements describe a function a system must be able to perform, including component, behavioral, and functional aspects. Quality and performance aspects, like throughput, reliability, and security, as well as constraining aspects, like physical or legal aspects, are summarized in non-functional requirements. While the derivation of these requirements is generally based on the analysis

¹⁶<https://www.bosch-smarthome.com/uk/en/>

¹⁷<https://shop.se.com/de/de/catalog/category/view/s/hemslogic/id/3252/>

¹⁸<https://openenergyplatform.org/>

¹⁹<https://openems.io/>

provided in the following section, we are also guided by our broad experience in numerous projects in cooperation with relevant industry partners, where prototypical EMSs for various scenarios have been developed and evaluated in large field tests, like e.g. MeRegio²⁰, C/sells²¹, flexQgrid²², and Smart East²³.

4.1. Analysis of Application Areas

Basis for the requirement elicitation process is an analysis of application areas, with which we aim to provide more context and information on the environment, in addition to Section 2. Domain knowledge and an understanding of the application's context is important for the quality of the requirements [Alebrahim et al., 2014, Antonelli et al., 2012]. Therefore, to aid in the formulation of requirements, this analysis defines the relevant application areas or domains [Loucopoulos and Champion, 1988], and is carried out by compiling typical applications for EMSs, as well as their characteristics and distinguishing factors. For this purpose, we consider the three application areas private households, commercial buildings, as well as districts and areas separately, and concisely illustrate the individual goals and system specifics. The latter, motivated by ISO 25010 [ISO/IEC, 2023], is achieved by pointing out functional, efficiency, compatibility, interaction, reliability and safety, security, as well as maintainability and flexibility aspects. Table 1 summarizes key differences and similarities between the application areas.

In private households, nearly two-thirds of the energy demand are used for space heating. They account for 27 % of the final energy demand in the EU, of which only 25 % is electricity [Eurostat, 2023], but this is expected to rise due to the ongoing electrification of heat and transport due to electric vehicles and heat pumps [Ruhnau et al., 2019]. Electricity generation by local PV plants is also growing rapidly, leading to an increase in households that produce parts of their electricity consumption themselves (often called "prosumers") [Sovacool et al., 2022, Kotilainen, 2019]. The usual goal is to optimize local PV usage and minimize the electricity needed from the public grid by using flexible devices, such as batteries, and shifting flexible electricity demand. In order to do this, private prosumer households often use EMSs to control their batteries, heat pumps, and/or charging processes [Zafar et al., 2020]. They can work rule-based [Berkes and Keshav, 2024], which in simple cases also produces optimal results, for instance when there is a flat electricity tariff by storing all excess PV production and discharging whenever there is a deficit, or use all sorts of optimization algorithms, including mixed integer linear programming, genetic algorithms, particle swarm optimization and more [Srilakshmi and Singh, 2022, Henggeler Antunes et al., 2022]. Home EMSs used in private households can either be provided as cloud services, e.g., by electricity providers, or operated locally on an edge device, e.g., a Raspberry Pi. In the latter option, EMSs can be operated based on open-source smart home systems like Home Assistant²⁴ or OpenHAB²⁵ which can be installed and used by everyone, but may be limited in terms of computing power. Hardware interoperability on the building level is a challenging task, due to a lack of standards and many vendor specific solutions. The provided user interfaces vary depending on the intended user group. Solutions like OpenHAB allow, for instance, the creation of own control rules, while others provide only simple visualizations. All functions should be provided without interruption, to ensure user comfort, but usually outages would only lead to loss of comfort for the affected household(s). Systems should be designed to avoid damage to devices and users.

²⁰<https://meregio.forschung.kit.edu/english/24.php>

²¹<https://www.wirsinteg.de/csells>

²²<https://flexqgrid.de/english/>

²³<https://smart-east-ka.de/>

²⁴<https://www.home-assistant.io/>

²⁵<https://www.openhab.org/>

Especially in private households, the limited computing power of local EMSs makes it favorable to outsource optimization, load prediction, or PV forecast to a cloud service. However, a strong argument for using local systems is the high level of privacy protection, as no data on electricity consumption, which can be used to draw conclusions about residents' behavior, has to be shared with cloud providers [Boiko et al., 2024]. Maintainability and (software) flexibility are crucial for EMS developers and providers, especially for offering their customers continued safe and secure systems and allowing support for more and new hardware.

In commercial buildings, energy management algorithms like e.g. proposed by [Chen et al., 2019, Ding et al., 2019, Oldewurtel et al., 2012], typically address the optimization of the Heating, Ventilation and Air Conditioning (HVAC) system, controlled centrally or for rooms individually. Usually, these buildings are equipped with a Building Automation System (BAS) on which a Rule Based Control (RBC) strategy is implemented. The latter is replaced with an optimization-based approach given an EMS is installed. The efficiency, both from a software and energy perspective, varies with the employed algorithms and depends on the local systems [Al-Ghaili et al., 2021]. Compatibility, like in the residential case, can be a challenge, however, with larger facilities and larger associated investments, customized integrations are more reasonable than in the residential case. In commercial buildings, the correct operation can be of critical importance for the organization utilizing the building. Therefore, the building optimization system might have to be executed on-premise to prevent outages caused by internet failures. Using cloud-based energy management systems in commercial buildings can also come with challenges regarding privacy and security [Anthi et al., 2018]. Commercial buildings might be utilized by organizations that are privacy-sensitive and thus do not permit data to be stored in the cloud. Other organizations, however, might be rather price-sensitive and hence prefer to use an optimization algorithm provided as cloud service while configuring the BAS to fall back to RBC in case of connection issues. From a maintainer and vendor perspective, again, maintenance and flexibility are important for the operation of existing systems and the further development of the product.

Districts and areas differ from those categories due to their size and, most importantly, the involvement of energy grids. One major reason to conduct energy management on the level of facilities, districts, and even on a regional scale is grid operation. With increasing decentralized generation, especially from renewable energy sources, and increasing demands from electrification, the need for monitoring the utilization of the grid and its power quality (see [Chawda et al., 2020]) and actively influencing energy flows to prevent or resolve undesired situations is rising (e.g., [Volk et al., 2017]). Energy management on an area level, therefore, often considers the associated energy grids, especially in the case of micro-grids. Another reason for area-level energy management is the optimal, e.g., cost-minimal, operation of all the generators, storage systems, and flexible loads in the area (e.g., [Roccotelli et al., 2022]). Control of the different flexible devices and/or buildings in the area can be achieved with more or less direct mechanisms, ranging from direct device access to indirect, highly aggregated control signals [Förderer et al., 2022]. The practical implementation and derived qualities like efficiency, reliability, and safety, depend on the selected orchestration mechanism, local regulation and the characteristics of the area in question, e.g., who owns the devices and energy grids and whether there are any fees for using the public grid in a given scenario. Since, in a region, there can be any number of commercial and residential buildings combined, the interoperability challenge is amplified manifold. Standardized interfaces, models, and EMSs for each building can alleviate this challenge [Khalid, 2024]. Users may be provided with user interfaces for checking the current regional status and history, or making inputs, such as electric vehicle charging settings. In districts and areas, privacy-sensitive data, e.g. of many households, may need to be protected, which can be done using data aggregation due to the larger scale [Kursawe et al., 2011, Varenhorst et al., 2024, Langer et al., 2013]. Here, a higher level of aggregation and abstraction is additionally beneficial to keep the amount of data that has

Table 1. *Comparison of three areas in which EMSs are used.*

	Private Households	Commercial Build-ings	Districts and Areas
Scale	Small	Medium	Large
Exemplary applications	Space heating, prosumers (PV), heat pumps, electric vehicles	HVAC systems, optionally PV and electric vehicle charging stations	Buildings and energy grids, especially electrical distribution grids
Optimization goals	Maximize self-consumption, minimize energy demand from the grid and optimize with respect to electricity prices	Optimization of HVAC systems to reduce energy demand while maintaining comfort	Cost-minimal operation of all generators and flexible loads and storages in the area
Information used for optimization	Building models, prediction of inflexible electricity demand and thermal energy demand, weather forecasts and local PV supply	Weather forecast and occupancy predictions, if sensing devices available	Non-standardized models due to higher level of aggregation and abstraction, current systems state, forecasting of energy-related time series
Implementation	On-premise with local electricity-saving edge devices, or cloud-based services	On-premise, to prevent outages, based on existing BAS with RBC strategy	Dedicated server due to scale, can be on-premise for factories
Privacy	Privacy-sensitive data, e.g., residents' behavior	Privacy-sensitive data possible – trade-off between privacy and price with cloud storage	Data aggregation can be used to protect privacy-sensitive information

to be managed and the computation times on an acceptable level, also resulting in a need for different models. Aggregation is especially important on higher grid and system levels. Smart energy-optimized areas may, for instance, aggregate their flexibility on the feeder level (e.g., [Volk et al., 2017]). In such a scenario, control signals need to be disaggregated for their implementation upon reception. Reliable and safe operation are especially important on the regional level, as faults may leave many buildings without energy. For achieving reliable and safe operation, maintainability and flexibility in software are especially helpful on this level, compared to the other two.

From a general perspective, the basic building blocks needed for energy management in all three application areas are very similar, that is, functionality for determining and assessing the current systems state, forecasting of energy-related time series, optimization of load schedules or similar control signals, and controllers implementing the schedules. The implementation, however, may vary due to the different properties and specific demands present in the application areas. In all three areas, privacy concerns need to be taken into account due to the presence of privacy-sensitive information.

4.2. Functional Requirements

With the analysis of application areas presented in the previous section and the described typical energy management applications in mind, we now derive functional requirements.

The first requirement directly results from the service and stakeholder concept discussed in Section 2. It is:

FR01: A service developer must be able to derive a functional service with the service framework from existing forecasting or optimization code.

Here, the intention of the service provider clearly is to allow EMSs to utilize the existing forecasting or optimization algorithm by interacting with the web API of the service²⁶. Hence the second and third requirements are:

FR02: An EMS must be able to interact with the service over a web API provided by the service.

FR03: An EMS must be able to request a forecast or optimized schedule utilizing the API of the service.

The forecasting or optimization algorithms wrapped by the service framework will usually require some form of input data. Considering a PV power generation forecast as example, this could be the global position and time of the target system. Furthermore, the data format returned by a forecasting or optimization algorithm will obviously be specific to it and should contain all the information the algorithm needs for processing the expected result. The latter includes constraints that should be obeyed by optimization algorithms. Therefore, the fourth requirement is:

FR04: A service developer must be able to specify the format of the input and output data exchanged due to an EMS request for a forecast or optimized schedule from the service API.

Some services may implement *system-specific parameters* that must be *fitted* utilizing historical measurements of the system subject to forecast or optimization as a prerequisite for high-quality results. Note that the algorithm for fitting the system-specific parameters is considered to be a part of the existing forecasting or optimization code. As services should be usable by a large number of EMSs, this fitting process should be manageable via the service API²⁷. Returning to the PV power generation forecast example, one could conceive that the forecasting code contains a small neural network that has been trained using the power generation of several PV systems, thus representing an average system. However, if power generation measurements of a specific PV system are available, it is possible to adapt (fit) the weights and bias terms of the neural network (*system-specific parameters*) such that the prediction error is minimized for the specific system. The fitting procedure is part of the service and the fitting process can be initiated by calling the respective API endpoint with the required input data, which would be a time series of historic power generation data for the PV power generation forecast example. Furthermore, we need to consider privacy-sensitive users, i.e. users that do not accept any of their data to be stored in a cloud database, which implies that it must be possible for EMSs to store the fitted parameters locally²⁸. Finally, we need to consider EMSs

²⁶Note that we formulate the remaining requirements about the derived service to improve readability. Later, in particular in Section 5.2, we discuss that the requirements need to be fulfilled by the service framework.

²⁷Note that web services exposing machine learning models usually implement a different approach, i.e. that the training will be carried out before deployment and hence that the parameters of the model are identical for all clients. In order to emphasize this difference, we explicitly refer to *fitting system-specific parameters* instead of *training models*. Furthermore, it is worth noting that our proposed approach will also work for foundation model based forecasting approaches, like e.g. TimeGPT introduced in Section 3.2. While such algorithms may or may not need to fit system-specific parameters for good performance, exposing them as services is nevertheless reasonable to enable a widespread application in EMSs.

²⁸One could argue that a user who opposes storing data in a cloud database will likely not want to use an EMS that utilizes forecasting or optimization services operated by an external service provider at all. On the other hand, service providers might guarantee that data exchanged with a service is deleted immediately after processing, which seems like a fair compromise between privacy and cost efficiency.

that are not capable of reliably storing historic recordings of measurements or fitted parameters on-premise, e.g. very likely a large fraction of EMSs operating in private households. While this seems like a major difference at first glance, it turns out that this scenario imposes no additional requirements for the service. The discussion behind this finding is out of scope at this point but can be found in Appendix A. The resulting requirements are thus:

FR05: An EMS must be able to fit system-specific parameters of a service utilizing its API.

FR06: A service developer must be able to specify the format of the input and output data exchanged while an EMS interacts with the API of a service to fit the system-specific parameters.

FR07: An EMS must have the option to store fitted system-specific parameters locally.

API calls made by an EMS may take a significant amount of time before the result becomes available. Consider e.g. the service providing PV power generation forecasts used as a running example for which fitting the system-specific parameters involves training a neural network that might take several minutes to hours. On the other hand, computing forecasts or optimized schedules might require a decent amount of time too, e.g. if computing an optimized schedule for a larger building involves solving a complex linear program. As such response times are different from typical values of web services, we formulate it as an additional requirement:

FR08: An EMS must be able to make calls to the API of the service which may take several hours to compute.

Finally, it is well known that *documentation* is important for the widespread adoption of APIs [Hunter, 2017, Jin et al., 2018]. Here documentation refers to the description of the functionality of a service, in particular its API and the data format for interactions with the latter. Furthermore, development efforts can be reduced by automatically generating the documentation from the corresponding source code, which is additionally beneficial as it prevents that changes in the code are not reflected in the documentation. The resulting final functional requirement is thus:

FR09: A service developer should be able to automatically generate a documentation for the API of a service.

4.3. Non-Functional Requirements

Extending the content above, this section presents non-functional requirements for the service framework. To this end, we first consider the envisioned target state that professional service providers operate services which are utilized by a large number of EMSs. Thus, the availability of these services is very likely of critical importance for the intended functioning of a large number of EMSs. This implies that service providers need to apply state-of-the-art computing cluster techniques for operation. Furthermore, the data exchanged between EMS and services may contain sensitive information and should thus be encrypted²⁹, especially as a large share of EMSs will likely communicate with services over the public internet. Finally, operating services may require significant compute resources and energy. A service provider may, hence, wish to restrict access to services to certain EMSs. This leads to the following requirements:

NFR01: A service provider must be able to operate services with high availability and scalability.

NFR02: An EMS must be able to communicate with the service over an encrypted connection.

²⁹Note that it is additionally reasonable to demand that the data exchanged between EMS and service cannot be altered in transit. However, we do not add this point as a separate requirement as it is automatically fulfilled if the communication is securely end-to-end encrypted, e.g. with HTTPS

NFR03: A service provider must be able to restrict access to a service to authorized EMSs.

As the correct functioning of the proposed service framework is substantial for the stable operation of the derived services, it becomes clear that service developers and providers must be convinced that the framework is implemented correctly to adopt it. Furthermore, service providers will likely not utilize the framework, if no reasonable maintenance concept exists, which suggests that future problems in the framework will be addressed and solved quickly.

NFR04: A service developer/provider should be able to validate the correct implementation of the service framework.

NFR05: A service developer/provider should be able to verify that the service framework is actively maintained.

Beyond the commercial aspect, an important intended application of the service framework is to empower academic researchers to derive functional services from existing forecasting or optimization code. For this, one needs to consider the limited resources typical for academic research, which implies that the task of deriving a service should require minimal effort³⁰. Furthermore, it should be regarded that some service developers might not have a strong expertise in applied informatics but should still be able to utilize the proposed framework. An example to illustrate this demand could be a project with public funding dedicated to energy management in private households carried out by a consortium of research groups. In such a case, it may appear beneficial to integrate a research group dedicated to energy meteorology to develop a forecast service for PV power generation without demanding that this group cares about the operation and implementation details of the service. This leads to the following requirements:

NFR06: A service developer should be able to derive a service with minimal effort from an existing forecasting or optimization algorithm.

NFR07: A service provider should be able to operate a service without requiring expert knowledge about IT infrastructure.

A service operating for a longer time may need continuous development work by both service developer and provider, e.g. in order to maintain or even improve performance and usability. Such efforts could include breaking changes, like e.g. an adaption of the data format which is not backward compatible. In order to give EMS developers time to adjust to those changes, it is common to operate an old and a new version in parallel. However, this implies that the EMS developer must be able to select which version of a service should be utilized, thus leading to the following requirement:

NFR08: An EMS developer must be able to specify which version of a service should be utilized.

While deriving FR09 above, we have argued that documentation is important for the adoption of APIs by EMS developers. However, beyond the pure existence of a documentation, it appears reasonable to demand that the latter should allow EMS developers to rapidly comprehend the API of a service. Furthermore, the main intention of a EMS developer reading the documentation is likely to implement a client in order to interact with the API of a service. For this, we demand minimal effort of implementation again, assuming it will likely support widespread adoption of the corresponding service. Hence, our final two requirements are:

NFR09: An EMS developer should be able to quickly understand the API of a service by utilizing the documentation.

NFR10: An EMS developer should be able to implement a client to interact with the API of a service with minimal effort.

³⁰This is obviously beneficial for service providers with a commercial background too.

We are convinced that the requirements derived in the present and previous section are a solid foundation for deriving a framework for provisioning forecasting and optimization algorithms as web services for EMSs, and demonstrate this suitability below.

5. Design Concept

Based on the requirements discussed above, we introduce the design concept of our proposed service framework in this section. To this end, we first present the API design, in particular as providing an API for forecasting and optimization code is the core functionality of our proposed solution. Based on the API design, we proceed to describe the internal operation of a service derived from the framework and finally conclude this section with a discussion about service operation.

5.1. API Design

As a first step, it is necessary to choose the paradigm on which the API of our proposed service framework should be based. We consider well-established approaches for web-based APIs (FR02). These are REST [Fielding, 2000], Remote Procedure Call (RPC) (in particular gRPC³¹) as well as GraphQL³². As all three candidates are generally suited to satisfy the functional requirements we focus on the non-functional requirements in order to select the best suited paradigm. The relevant requirements are understandability (NFR09) as well as ease of client implementation (NFR10). It is generally perceived that REST is the most favorable approach regarding these demands [Hunter, 2017, Jin et al., 2018], which is therefore selected.

As a next step, we define the functionality of the API that is provided by the service framework. The selection of REST implies that all communication between client and service will use the Hypertext Transfer Protocol (HTTP) and that the functionality must be mapped to Uniform Resource Locators (URLs). It should be noted that we will only note down the relative part of URLs for the sake of brevity and to highlight that the domain is not relevant for the structure of the API, i.e. we use `/endpoint1/` instead of the full notation `https://some-service.example.com/endpoint1/`. The selection of REST furthermore implies that we need to define which HTTP method (like e.g. GET, POST, PUT, or DELETE) must be used in order to receive a desired outcome while interacting with a specific URL. We will henceforth refer to the combination of HTTP method and (relative) URL as *API method*. Further introduction about web-based communication over HTTP can be found in the usual introductory texts or as a short summary in [Jin et al., 2018].

FR03 dictates that EMSs must be able to retrieve a forecast or optimized schedule from a service. Furthermore, we need to consider that a service may take minutes or even hours to compute the result (FR08). As especially the latter is far beyond typical timeouts of HTTP servers³³ it is infeasible to directly return the computation result. Instead, we define three API methods to overcome this issue:

```
POST  /{version}/request/
GET   /{version}/request/{task_ID}/status/
GET   /{version}/request/{task_ID}/result/
```

The intended interaction of an EMS with these API methods is as follows:

1. The EMS issues a POST call to the `/ {version}/request/` endpoint containing the required input data (see FR04). Note that `{version}` is a placeholder that must be filled with the desired version of the targeted service, which is required to satisfy NFR08, and

³¹<https://grpc.io/>

³²<https://graphql.org/>

³³The default timeout of nginx for a read operation is, for example, 60 seconds.

could e.g. have a value of `v2`, see the example provided below. The service checks whether the input data is correct. If that is the case the service starts computing the result in the background and returns an ID, e.g. 123, associated to this task (more details about the latter are provided in the following two sections).

2. Using the ID of the request, the EMS should issue calls to the endpoint: `GET /{version}/request/{task_ID}/status/`³⁴. Note that `{task_ID}` is a placeholder too. Regarding the example above, the endpoint would be `/v2/request/123/status/`. For each call, the service will compute and return the status of the computation, that is one of `queued`, `running`, or `ready`³⁵.
3. Once the service has finished processing the result and a `ready` status has been observed, the EMS can issue a `GET` call to the `/{version}/request/{task_ID}/result/` endpoint to retrieve the output of the computation, i.e. the forecast or optimized schedule.

Additionally, to the ones defined above, it is necessary to specify API methods to allow fitting system-specific parameters of a service in order to satisfy FR05. As potentially long processing times (FR08), as well as versioning (NFR08), need to be considered again, it appears reasonable to take over the concept introduced above and define the respective API methods as:

```
POST /{version}/fit-parameters/
GET  /{version}/fit-parameters/{task_ID}/status/
GET  /{version}/fit-parameters/{task_ID}/result/
```

The intended usage of the `/fit-parameters/` endpoints is equivalent to the pattern discussed for `/request/` above.

Finally, and in order to satisfy NFR03, we need to consider authorization, which implies authentication, to allow service providers to restrict access to specific clients. Following [Späth, 2023, Saeed and Abdallah, 2022, Kornienko et al., 2021] the currently best practice for REST APIs is token-based³⁶ authentication, in particular the utilization of JSON Web Tokens (JWTs). JWTs, as defined in [Jones et al., 2015], are special tokens that can be cryptographically validated. Regarding the scope of this work, JWTs are issued by a dedicated identity provider (see Section 5.4) to the client software. The signature of the token allows the service to check locally³⁷ whether a request of a client should be granted or not. Further details about the application of JWTs for web security are given in [Saeed and Abdallah, 2022].

5.2. Service Components

The concept described in this section arises from the requirements FR01 and NFR06, i.e. that it should be possible to derive a functional service from an existing forecasting or optimization code with minimal effort. Especially the latter (NFR06) imposes that as much functionality as possible should be realized by the service framework in order to keep the implementation effort for the service developer low. To this end we define a functional service to consist of several components that can be grouped into three categories as summarized in Figure 4:

1. Base: Containing the components necessary for executing the code of the service.
2. Service Framework: Containing all components generic to all services.

³⁴Note that this polling mechanism seems a bit inelegant at first glance. However, alternatives have severe downsides too, e.g. WebHooks imply the need for the EMS to be exposed on the network while WebSockets don't integrate well into documentation and tooling of REST APIs. See Chapter 2 in [Jin et al., 2018] for a more detailed discussion.

³⁵Note that we have not added a failed state as `ready` just implies that the result endpoint can be called. The information whether the requested computation has succeeded or failed is provided by the HTTP status code returned while calling the `/request/{task_ID}/result/` endpoint. This is the standard approach for communication over HTTP.

³⁶A token is considered as a string that is sent with every HTTP request to the webserver (the service in our case), most commonly in the HTTP header. The token is usually specific for each client and allows the webserver to validate if the request is permitted or not, e.g. by looking up the permissions associated with the particular token in a database.

³⁷Locally means here that the token can be validated without the lookup operation mentioned in the previous footnote, which supports scalability.

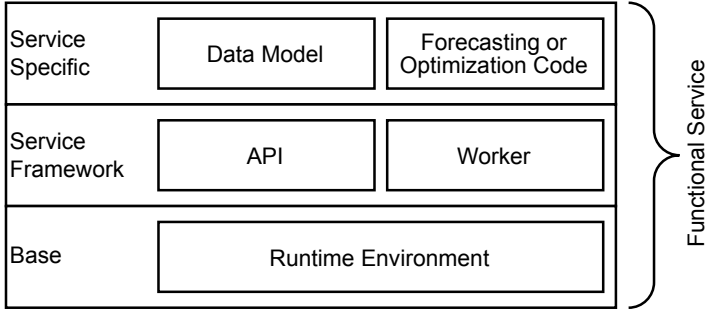


Figure 4. *Components of a service derived with the service framework.*

3. **Service Specific:** Containing all components a service provider must implement to derive a functional service.

The primary component of the service framework is the [API](#) as defined above. Additionally, we need to consider that computing the result of a request (as well as fitting system-specific parameters) might take significant time ([FR08](#)), while the [REST API](#) of the service should respond immediately. Hence, it is necessary to decouple the [API](#) from the interaction with forecasting or optimization code. Therefore, we introduce the *worker*, which is a second component provided by the service framework that is executed in a dedicated process and that is responsible for computing the requested results. It is worth noting that this concurrent processing is crucially important, as, otherwise, executing a forecasting or optimization code might block the [API](#) from responding to other calls from clients³⁸.

The service specific category contains the actual payload of the service, i.e. the forecasting or optimization code. Additionally, the requirements [FR04](#) and [FR06](#) need to be considered, i.e. that service developers must be able to specify the format of the input data for calls to `/request/` and `/fit-parameters/` as well as the output format returned by the corresponding `/result/` [API](#) methods. We will refer to the part of the implementation that defines these formats as *data model*³⁹. At this point, we will not further specify possible characteristics of the data model as the latter are tightly connected to the implementation of the [API](#). However, we will proceed discussing this topic in Section 6.2.

5.3. Service Architecture

It was discussed in the previous section (5.2) that a functional service must contain an [API](#) as well as a worker component and that these should be operated in distinct processes for performance reasons. It should be noted at this point that the service specific components, i.e. the data model as well as the forecasting or optimization code, are perceived to be parts of the [API](#) and worker components. The service specific components are consequently not explicitly mentioned in this section to promote readability.

Following from the execution of service components in distinct processes the necessity arises to establish some form of inter-process communication to allow services to operate. Considering the simplest case, i.e. a service consisting of two processes, one for the [API](#) and one for the worker, communication between the two could be established quite simply using queues. However, we need to consider [NFR01](#), i.e. that service providers should be able to operate services

³⁸Decoupling the [API](#) from computing results furthermore allows more sophisticated load management, like queuing tasks until resources for computation are available, while still allowing the [API](#) to be responsive to client calls.

³⁹We chose *data model* to be consistent with the terminology used by the framework employed for implementing the [API](#) component (i.e. FastAPI, see Section 6.2 below) which uses *model*. However, the latter collides with a potential model used in the forecasting or optimization code. Hence, we use *data model*.

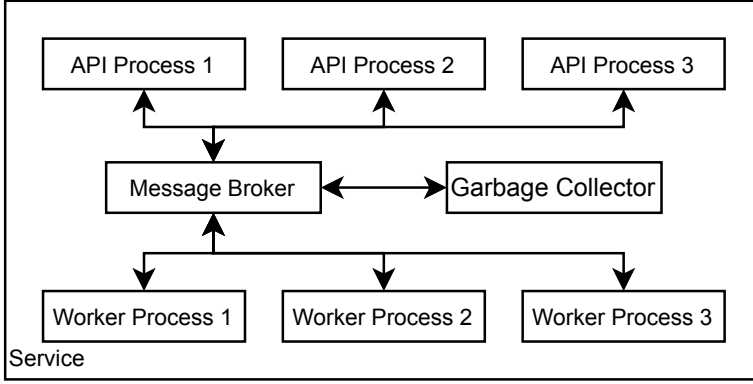


Figure 5. Internal architecture of a service with processes of components and communication between processes.

with high availability and scalability. From the latter follows that services might consist of multiple instances of [API](#) and worker components, while high availability imposes that the service components might be distributed over several machines. The usual approach in such a scenario, which is utilized in this work, too, is to leverage a *message broker* for communication between components. The resulting communication pattern within a service would be as follows:

- Every valid POST call to a `/request/` or `/fit-parameters/` endpoint should lead to the creation of a *task*, an object carrying the necessary information for computing the result, by the [API](#). The latter should assign an ID to the task and publish the task on the message broker to invoke the workers. Finally, the [API](#) should return the task ID to the client.
- A worker process should fetch the task from the message broker and start computing the result by invoking the forecasting or optimization code. Additionally, the worker should regularly publish status updates on the processing progress on the broker.
- In case of a call to a `/status/` endpoint, the [API](#) should fetch the latest status information about the corresponding task from the broker and return this information to the client.
- If a `/result/` endpoint is called, the [API](#) should fetch the result from the broker and return the result to the client.

Finally, it must be considered that a result that is fetched from the broker should not be deleted on the broker immediately. For example, consider that the communication between client and [API](#) might be interrupted, in which case the client will likely retry to fetch the result. However, in order to prevent unlimitedly growing memory consumption of the message broker, it is required to operate an additional component, i.e. the *garbage collector*. The duty of the latter is to delete the task-related data from the broker that are likely not to be required anymore.

The resulting internal architecture of a service, including communication channels, is indicated in Figure 5. It should be appreciated that the service architecture introduced above is strictly designed to support scalability and high availability, in particular, as subsequent calls from clients are not required to hit the same [API](#) process again. That is, it is not required that a call to the `/status/` endpoint will be handled by the same [API](#) process that created the corresponding task, as all task-related information is shared on the broker. Furthermore, it is possible to scale the number of [API](#) as well as worker processes to the actual load induced by clients.

5.4. Operation Concept

Building up on our considerations introduced above, we will conclude the service design in this section by discussing the missing piece, which is the operation concept. To this end, it should be recalled first that (in this work) a service consists of several components (like **API** or worker) and that in reality operation might require that several instances of each component are executed in parallel, possibly distributed over several machines (see Section 5.3). We have referred to the latter as processes in order to distinguish the running instance from the implementation. The management of these processes, commonly referred to as orchestration, is a tedious activity best left to specialized and well-established applications. In consequence, the first step towards the operation concept is to select an appropriate orchestration software, where it needs to be considered that service providers have different demands here, ranging from cloud computing professionals of commercial companies with a primary interest in high availability and scalability (NFR01) to academic researchers with no experience with the utilization of orchestration software (NFR07). Systematic comparisons of orchestration applications [Malviya and Dwivedi, 2022, Jawarneh et al., 2019, Mercl and Pavlik, 2019] suggest Kubernetes⁴⁰ for the first group and Docker Swarm⁴¹ for the second. Both orchestrators require that the processes of the service are wrapped in containers. As Docker⁴² containers are supported by both they are selected.

Beyond the execution of the service components as described in the previous sections, provisioning of functional services to **EMSs** requires that a service provider operates additional supportive applications. The need for the first one of these supportive applications arises from the simple demand that the **API** containers must be accessible for the **EMSs** (FR04 and FR06), as well as that requests from clients should be distributed over the **API** containers of a service, in order to allow highly available and scalable operation (NFR01). We will refer to the application that fulfills this duty as *gateway*. However, it should be noted that such applications are often alternatively named *reverse proxy* or *ingress*⁴³, the latter in particular in the context of Kubernetes. It is common, too, that the gateway encrypts the communication with the client, i.e. by utilizing the **Hypertext Transfer Protocol Secure** (HTTPS), on behalf of the payload application, in our case, the service. This procedure removes the burden of integrating the security-relevant and vastly complex encryption logic into the code base of the **API** component while still satisfying the requirement that communication between client and service must be secure (NFR02). The second supportive application is necessary to issue **JWT** tokens to clients in order to satisfy NFR03 as discussed in Section 5.1. Such applications are usually referred to as **Identity Provider** (IdP) and the current best practice choice for issuing **JWTs** is the **OpenID Connect** (OIDC) protocol [Scott and Neray, 2021, Kornienko et al., 2021]. Hence, our operation concept follows this best practice and intends that a well-established IdP software, like e.g. Keycloak⁴⁴, is used in order to administrate access control and to issue **JWT** tokens to the client software using **OIDC**. Finally, it is worth noting that it may not be necessary that every service provider operates an IdP by themselves. Especially in the context of academic research, it is likely sufficient that one partner operates an IdP, e.g. for a project, while services can still be distributed over several partners. This significantly reduces the effort for providing services for those partners that do not serve the IdP, as operating the latter is a complex task requiring specialized knowledge. This possibility should support especially academic researchers with backgrounds other than informatics.

⁴⁰<https://kubernetes.io/>

⁴¹<https://docs.docker.com/engine/swarm/>

⁴²<https://www.docker.com/resources/what-container/>

⁴³<https://kubernetes.io/docs/concepts/services-networking/ingress/>

⁴⁴<https://www.keycloak.org/>

6. Implementation

In this section, we present the reference implementation of the design concept introduced above, which is released alongside this paper as an open source repository and which we refer to as **Energy Service Generics (ESG)**. The latter is implemented in Python, which has been selected due to the availability of many relevant software libraries for the implementation of forecasting and optimization algorithms, like e.g. PyTorch⁴⁵, TensorFlow⁴⁶ or Pyomo⁴⁷. Furthermore, Python offers high connectivity to other programming languages, especially the integration of C and C++ is natively supported⁴⁸. Furthermore, the latter allows the integration of code written in other programming languages that can be compiled to C, e.g. go⁴⁹. While there may be no simple solution to integrate code written in some programming languages, like e.g. especially Java, into the **ESG** framework, the concepts described in Section 5 are sufficiently generic to allow an implementation of a similar framework in other programming languages. However, this is not subject to this work, which presents only one implementation of the design concept, which is the **ESG** framework written in Python.

It is worth noting that the following sections are, on purpose, rather concise, i.e. cover the relevant points to demonstrate that the proposed implementation satisfies the respective requirements. Additional details will be omitted for the sake of conciseness, in particular as the implementation should be continuously developed further as an open source project (see Section 7), which will likely lead to significant differences compared to the code state at the publication date of this paper. However, further information is provided in the **ESG** repository⁵⁰, including examples how services and clients can be implemented.

6.1. Worker, Garbage Collector, and Inter-Process Communication

In order to ensure that our implementation is robust and to minimize future maintenance effort, we refrain from developing a custom approach for the inter-process communication between **API** and worker. Instead, we utilize Celery⁵¹, a well-established, stable, and open-source Python library for distributed task execution. It is worth noting that Celery is particularly well suited for use cases similar to ours, i.e. to decouple client-facing web components from worker processes in a scalable fashion. Furthermore, Celery supports multiple message brokers, including the well-established Redis⁵² and RabbitMQ⁵³, which gives service providers flexibility to choose a broker matching their demands, tech stack, and/or skill set of employees. We provide an implementation of a generic worker, which invokes the service specific forecasting or optimization code in the **ESG** package. The worker makes use of Celery to interact with the message broker and implements the functionality described in Section 5.3. Finally, Celery provides a garbage collector functionality. Dependent on the choice of message broker the latter may be available without the need for a dedicated process, more details are provided in the respective part of the documentation⁵⁴.

⁴⁵<https://pytorch.org/>

⁴⁶<https://www.tensorflow.org/>

⁴⁷<https://www.pyomo.org/>

⁴⁸<https://docs.python.org/3/extending/extending.html>

⁴⁹<https://pkg.go.dev/cmd/cgo>

⁵⁰<https://github.com/fzi-forschungszentrum-informatik/energy-service-generics/>

⁵¹<https://docs.celeryq.dev/>

⁵²<https://redis.io/>

⁵³<https://rabbitmq.com/>

⁵⁴<https://docs.celeryq.dev/en/stable/userguide/configuration.html#result-expires>

6.2. API

The first step towards implementing the **API** is to select an appropriate framework to build upon. Considering the choice of Python as programming language, it follows that two well-established web frameworks are available⁵⁵, these are Flask⁵⁶ and FastAPI⁵⁷. Several requirements need to be considered in order to select one of the two candidates objectively. In particular, these are **FR09** (service developers should be able to automatically generate a documentation), **NFR09** (EMS developers should be able to quickly understand the **API** using the documentation) as well as **NFR10** (EMS developers should be able to implement a client with little effort). We have selected FastAPI because it is generally considered to have a better integration of the OpenAPI specification⁵⁸ and tool ecosystem [Singh, 2023, Krebs and Cruz Martinez, 2022]. OpenAPI schema is a gold standard for documenting **REST APIs**. FastAPI is capable of automatically generating and hosting OpenAPI schema documents as well as serving Swagger UI⁵⁹. The latter is an interactive **API** documentation building up on the former, thus satisfying **FR09**. Furthermore, an interactive documentation is considered to be especially helpful for client developers to quickly understand an **API** [Hunter, 2017], which makes FastAPI a reasonable choice to fulfill **NFR09**. Another helpful tool from the OpenAPI ecosystem is Swagger Codegen⁶⁰ which allows the automatic generation of large shares of client code for an impressive number of programming languages, thus satisfying **NFR10**.

Using FastAPI, our implementation of the **API** component is capable of serving the endpoints defined in Section 5.1. It is fully functional, apart from the definition of the data models, i.e. the input data for **POST /request/** and **POST /fit-parameters/** as well as the corresponding **GET /result/** **API** methods, which need to be defined by the service provider. It is worth noting that the definition of the data models is simple and fast to implement in order to fulfill **NFR06**. A practical example and additional details are provided in Section 8.1.2. Furthermore, our implementation of the **API** component puts the communication concept defined in Section 5.2 into action, i.e. it transposes the **HTTP** calls of the clients into interactions with the message broker using the Celery framework introduced above, but only after the corresponding **JWT** of the call has been verified and checked. We utilize the PyJWT⁶¹ package for the latter. Finally, it is worth mentioning that our implementation solely utilizes the **JavaScript Object Notation (JSON)** data format for all data exchange, as the latter is interpretable for humans and machines alike and furthermore considered to be the best choice for **REST APIs** [Hunter, 2017].

6.3. Other Functionality

Besides the implementation of the **API** and the process management system, the **ESG** package provides additional useful functionality for the implementation of and interaction with services. Particularly relevant is a generic client that can be used to trigger calls to services from Python source code. Furthermore, the package contains building blocks for data models, in order to reduce the effort for implementing these as well as to prevent code redundancy, see Section 8.1.2 for further details. Additionally, useful utility functions are available, like e.g. to parse pandas⁶² "DataFrames" from **JSON** data.

⁵⁵Django is a well-established and Python-based web framework too. However, Django is primarily intended for the development of websites and heavily relies on communication with a database, and is hence not regarded a suitable candidate here.

⁵⁶<https://palletsprojects.com/p/flask/>

⁵⁷<https://fastapi.tiangolo.com/>

⁵⁸<https://swagger.io/specification/>

⁵⁹<https://swagger.io/tools/swagger-ui/>

⁶⁰<https://swagger.io/tools/swagger-codegen/>

⁶¹<https://pyjwt.readthedocs.io/en/stable/>

⁶²A popular Python library for analysis and manipulation of time series data.

7. Community Concept

Above, we have discussed the design and implementation of the service framework that satisfies the requirements presented in Section 4.2 and 4.3, with two exceptions: So far, we have not addressed that service developers and providers should be able to verify the correct implementation (NFR04) and that the service framework should be actively maintained (NFR05). We satisfy the first of these two requirements by releasing our implementation (i.e. the *ESG* package) as free and open-source repository, including an extensive documentation. A link to the repository is provided in the section "*Supplementary Material*" at the end of this paper. In order to fulfill NFR05, we strive to find a concept that ensures the continuous maintenance of the service framework. To that end, we first consider that typical reasons for modern open source projects to fail are usurpation by competitors, the project being not useful anymore or the lack of time and interest of developers [Coelho and Valente, 2017]. While we certainly cannot prevent that forecasting and optimization services may be not useful anymore at some point in the future, we strive to anticipate the remaining reasons leading to the abandoning of open source projects by initiating the *Open Energy Services* community. Following the classification and discussion about possible organizational forms of communities that maintain open source projects in [Eckert et al., 2019], we chose the form of an autonomous community, i.e. one that is not closely tied to nor owned by a company or organization, as we want to avert that service providers may object utilizing the framework due to interest clashes with the former. Furthermore, we have considered current best practices [Fogel, 2022, Mateos-Garcia and Steinmueller, 2008] while setting up the internal structures of the community in order to let decentralized and vital processes for exchange, development, and maintenance flourish.

The efforts of our community are not to be limited to the maintenance of the framework. Instead, the goal is to directly support the widespread adoption of *EMS* by connecting service developers, service providers and *EMS* developers, all of which are invited to become members of the community. We strive to develop a number of forecasting and optimization services and release these as open source repositories. Furthermore, it is intended to operate a selection of services for the public⁶³. Here, the concept is to first provide basic building blocks for energy optimization, like e.g. forecasting services for electric loads or *PV* power generation, in order to bootstrap energy management-related research and development activities, particularly by further academic institutions. The main communication medium is the community website⁶⁴, where we will release further information about recent activities and developments arising from our continuous efforts. Furthermore, and inspired by the FAIR principles for research software [Lamprecht et al., 2020], the website contains a registry for service-related open source repositories, as well as a registry for operated services. The main intention is that the community website should become the primary source of objective information about forecasting and optimization services for energy management applications in the future.

The founding members of the *Open Energy Services* community are the research institutions to which the authors of this paper belong. However, we explicitly invite other institutions and persons to join in order to establish a solid basis for the development and operation of forecasting and optimization services that enable energy management solutions at scale.

⁶³Note that we plan to provide these services free of charge. However, we will likely take measures to limit the load to our compute resources, e.g. by restricting access to registered clients or by limiting the number of *API* calls allowed per time.

⁶⁴<https://open-energy-services.org/>

8. Evaluation

The goal of this paper is to support the widespread adoption of EMSs in order to unlock the flexibility and energy savings potentials of end consumers. To this end, we contribute a concept for a software framework that allows the derivation of fully functional services from existing forecasting or optimization code with ease. Furthermore, we publish an open-source implementation of our proposed approach. Additionally, we establish a community in order to ensure the future maintenance of the framework, but also to support the widespread adoption of forecasting and optimization services in energy management applications.

We demonstrate in this section that our contributions are actually sufficient to reach our claimed goal of this paper, i.e. that our framework and community concept are useful for establishing forecasting and optimization services for energy management applications. To this end we begin by showcasing that our framework allows service developers to derive forecasting or optimization services with ease by providing a practical example for the implementation of a service. As it is also intended that the derived services can be integrated effortlessly into EMSs, we demonstrate the latter by providing two practical examples of client scripts that allow interacting with the previously derived example service. We continue our practical demonstration by exploring the scalability of services in order to verify that the services derived with our framework are indeed capable of serving thousands of EMSs. We conclude our evaluation with a structured comparison of the derived requirements with our concept and implementation, to illustrate the completeness of our work.

8.1. Deriving Services

As a first step towards proving the relevance of this work, we demonstrate how a simple (but fully functional) PV power generation forecast service can be implemented using the proposed framework. The following subsections present and discuss the implementation of the components of the service. It should be noted that the forecasting algorithm of the illustrated service, including the quality of the forecasts the service could compute, is not of particular interest. Instead, the aim of this section is to demonstrate the value of the framework, which can be perceived by realizing how short the code listings below are. Besides the forecasting or optimization algorithm, which must be implemented anyway, deriving the functional example service requires less than 100 lines of code. On the other hand, at the time of writing, the ESG repository contained about 2400 lines of code, excluding examples and documentation. It is thus obvious that utilization of the ESG framework is much more work-efficient compared to the implementation of services from scratch, as the latter would require the developer to implement a larger fraction of the code provided by the framework. In fact, implementing services from scratch might be far more time-consuming than the proportion of lines of code suggests. This is, in particular, the case as the code using the framework is quite simple, as demonstrated below, while some parts provided by the framework are complex and need to be implemented carefully, e.g. due to security concerns or impacts on runtime behavior.

It is worth noting that the code listings provided below are part of the ESG repository⁶⁵, whereby the online version might be adapted to any potential future changes of the ESG framework. We thus suggest all those wishing to reproduce this example, i.e. run the example service, to use the files provided in the repository.

8.1.1. The Forecasting or Optimization Code

The forecasting or optimization code is the payload of the service, i.e. the goal of applying the ESG framework is to make this component accessible to EMSs. The ESG framework has

⁶⁵https://github.com/fzi-forschungszentrum-informatik/energy-service-generics/tree/main/docs/examples/basic_example

been designed to make integration of existing forecasting or optimization code simple, which is demonstrated in the example by utilizing the popular `pvlb`⁶⁶ for computing the PV power production forecast. However, it is worth noting that the framework does not induce any restrictions on the forecasting or optimization code wrapped by it. For example, linear programs, classical statistical models, or fully black-box machine learning approaches are all possible. Even model ensembles can be realized, either as a single service or as a service that calls other services as ensemble members.

In order to allow `pvlb` to compute the forecasts, it is necessary to provide the corresponding input data to the library. The first part of this input data is the specification of the PV system for which the forecast should be computed. We assume that the PV system is sufficiently described by the geographic position, i.e. latitude and longitude, as well as the geometry of the PV system, i.e. azimuth and inclination, and the peak power. All other options to describe the PV system offered by `pvlb`⁶⁷ are neglected to make this example not more complex than necessary. The second part of the required input to compute PV power prediction consists of meteorological forecast data, especially forecasts of solar irradiance.

Before we proceed with the discussion about the handling of input data by the ESG framework, it is worth recalling that the framework supports two types of endpoints, these are `/request/` and `/fit-parameters/`. This differentiation arises from the requirement FR05, i.e. that the ESG framework should allow forecasting or optimization code that uses ML approaches, which induces that some parameters of a model must be fitted utilizing observations of the target system, see Section 5.1 for details. Regarding the example above, we assume that the service is intended to produce PV power generation forecasts for systems for which geometry and peak power values may be unknown and need to be estimated from power production measurements. The parameter fitting has been implemented with a simple least squares approach, although it should be noted that this choice has no particular relevance for the present example. Thus, the input data necessary to obtain a forecast is separated into two groups: latitude and longitude are *arguments* while azimuth, inclination, and peak power are *parameters*. It is worth noting that parameters are not necessarily interpretable as in this example, e.g. weights and bias terms of a neural network could be parameters too. Finally, it should be considered that it may not be reasonable to demand all input data as client input. In the present example, the service fetches the meteorological data automatically from a third-party web service, which would, in practice, make the interaction with the service more convenient and less error-prone for the client. Finally, the following points concerning the code listing below should be noted:

1. The format of `input_data` and `output_data` is implicitly defined in the corresponding data models, which are introduced in the following section.
2. The functions `predict_pv_power`, `fetch_meteo_data` as well as `fit_with_least_squares` have been omitted from the listing, as the practical implementation details of those are not of particular interest for the scope of this work. However, the code of the omitted functions can be found in the repository of the ESG framework.
3. Implementing `fit_parameters` is optional and can be omitted for services without fittable parameters. An example without fittable parameters could be a service wrapping the AutoPV algorithm proposed in [Meisenbacher et al., 2023].

Listing 1. *Integration of the forecasting or optimization code.*

```
from esg.utils.pandas import series_from_value_message_list
from esg.utils.pandas import value_message_list_from_series
```

⁶⁶<https://github.com/pvlb/pvlb-python>

⁶⁷https://pvlb-python.readthedocs.io/en/stable/user_guide/modelchain.html

```

def handle_request(input_data):
    arguments = input_data.arguments
    parameters = input_data.parameters
    meteo_data = fetch_meteo_data(
        lat=arguments.geographic_position.latitude,
        lon=arguments.geographic_position.longitude,
    )
    pv_power = predict_pv_power(
        lat=arguments.geographic_position.latitude,
        lon=arguments.geographic_position.longitude,
        azimuth=parameters.pv_system.azimuth_angle,
        inclination=parameters.pv_system.inclination_angle,
        peak_power=parameters.pv_system.nominal_power,
        meteo_data=meteo_data,
    )
    output_data = {"power_prediction": value_message_list_from_series(pv_power)}
    return output_data

def fit_parameters(input_data):
    arguments = input_data.arguments
    measured_power = series_from_value_message_list(input_data.observations.measured_power)
    meteo_data = fetch_meteo_data(
        lat=arguments.geographic_position.latitude,
        lon=arguments.geographic_position.longitude,
        past_days=90,
    )
    measured_power = series_from_value_message_list(input_data.observations.measured_power)
    fitted_pv_system = fit_with_least_squares(
        lat=arguments.geographic_position.latitude,
        lon=arguments.geographic_position.longitude,
        meteo_data=meteo_data,
        measured_power=measured_power,
    )
    return fitted_pv_system

```

It should be noted that the two functions defined above, i.e. `handle_request` and `fit_parameters`, are the only part of the implementation of the service that actually interacts with the forecasting or optimization algorithm. The duty of these functions is to correctly invoke the forecasting or optimization code with the necessary input data.

8.1.2. The Data Model

In addition to the forecasting or optimization code introduced above, the data model is the second component that is service specific and which must thus be defined by the service developer. The data models define the format of the data the client exchanges with the service. For a service without fittable parameters, i.e. a service with `/request/` endpoints only, it is sufficient to define the arguments required for computing the request as well as the result of the computation. The corresponding data models are called `RequestArguments` and `RequestOutput`. It is worth noting that `RequestArguments` could also contain constraints that optimization services should account for. Furthermore, `RequestArguments` will likely often contain a field that defines the temporal resolution of the generated forecast or optimized schedule. Alternatively, some services may implement a fixed temporal resolution that cannot be manipulated by the client. In the present example, the `PV` power generation forecast is always returned on an interval of 15 minutes.

In the case of a service with fittable parameters, it is additionally necessary to define the data format for the input and output data for the `/fit-parameters/` endpoints. The data models specifying the input for the fitting process are referred to as `FitParameterArguments` and `Observations`, and the corresponding output is `FittedParameters`. As the simple `PV` power generation forecast service used as an example is designed to provide functionality to fit parameters, it is necessary to define all five data models introduced above. The corresponding implementation is shown in Listing 2.

Listing 2. *Definition of the data models.*

```
from esg.models.base import _BaseModel
from esg.models.datapoint import ValueMessageList
from esg.models.metadata import GeographicPosition, PVSysystem
from pydantic import Field

class RequestArguments(_BaseModel):
    geographic_position: GeographicPosition

class RequestOutput(_BaseModel):
    power_prediction: ValueMessageList = Field(description="Prediction of power production in W")

class FitParameterArguments(_BaseModel):
    geographic_position: GeographicPosition

class Observations(_BaseModel):
    measured_power: ValueMessageList = Field(description="Measured power production in W")

class FittedParameters(_BaseModel):
    pv_system: PVSysystem
```

At this point, it is worth noting that the `ESG` package provides ready-to-use building blocks for data models. For example, in the code above, `GeographicPosition` is imported from `ESG`. The former is a data model too, which defines that a geographic position consists of latitude and longitude. Furthermore, it is crucial to note that the data models serve additional functionality beyond the definition of the data format. Particularly important is the provisioning of documentation of the format in human-readable form (e.g. the `description` in the example above) as well as defining permitted ranges for values, the latter is utilized by the derived service to automatically validate the input data provided by clients. An example for such a permitted range could be to enforce that values for latitude must be in the range of $-90^\circ \dots 90^\circ$. This rule is, in fact, implemented in `GeographicPosition`, although this is not directly visible in the code example above. However, it can be perceived by inspecting the source code of the `GeographicPosition` data model⁶⁸.

8.1.3. The Worker

The worker component is responsible for executing the tasks, i.e. computing requests or fitting parameters by invoking the forecasting or optimization code, as well as task scheduling. Further details are provided in Sections 5.2 and 5.3. The `ESG` framework utilizes the Celery library for implementing the worker, but extends the latter with functionality to make the implementation of services more convenient, for example by utilizing the data models for de-/serialization of input and output data. Thus, the main objective for implementing a worker is to wire up the data models with the forecasting or optimization code, which should usually require a rather simple program, as displayed in Listing 3, for the `PV` power generation forecast example service.

⁶⁸<https://github.com/fzi-forschungszentrum-informatik/energy-service-generics/blob/main/source/esg/models/metadata.py>

Listing 3. *Definition of the worker tasks.*

```

from esg.service.worker import celery_app_from_environ
from esg.service.worker import invoke_fit_parameters
from esg.service.worker import invoke_handle_request

from data_model import RequestArguments, RequestOutput
from data_model import FittedParameters, Observations
from data_model import FitParameterArguments
from fooc import fit_parameters, handle_request

app = celery_app_from_environ()

@app.task
def request_task(input_data_json):
    return invoke_handle_request(
        input_data_json=input_data_json,
        RequestArguments=RequestArguments,
        FittedParameters=FittedParameters,
        handle_request_function=handle_request,
        RequestOutput=RequestOutput,
    )

@app.task
def fit_parameters_task(input_data_json):
    return invoke_fit_parameters(
        input_data_json=input_data_json,
        FitParameterArguments=FitParameterArguments,
        Observations=Observations,
        fit_parameters_function=fit_parameters,
        FittedParameters=FittedParameters,
    )

```

8.1.4. The API

The [API](#) component connects the worker with the client by allowing the latter to trigger the computation of requests or fitting of parameters as well as retrieving the corresponding results. To this end, the [API](#) component has to check the client input for validity and create the computation tasks. Furthermore, the [API](#) component handles authentication and authorization of clients. More details about the [API](#) design are provided in Section 5.1.

The implementation of the [API](#) component is available ready-to-use in the [ESG](#) framework. However, in order to operate the [API](#) it is necessary, similar to the worker, to wire up the [API](#) with the other components, in particular with the data model and the worker. Furthermore, some information like name and version number must be provided too. Nevertheless, the necessary code to instantiate an [API](#) component is trivially simple and shown in Listing 4.

Listing 4. *Instantiation of the [API](#) component.*

```

from data_model import RequestArguments, RequestOutput
from data_model import FittedParameters, Observations
from data_model import FitParameterArguments
from worker import request_task, fit_parameters_task

api = API(
    RequestArguments=RequestArguments,
    RequestOutput=RequestOutput,

```

```

    FittedParameters=FittedParameters,
    Observations=Observations,
    FitParameterArguments=FitParameterArguments,
    request_task=request_task,
    fit_parameters_task=fit_parameters_task,
    title="PV Power Prediction Example Service",
)

if __name__ == "__main__":
    api.run()

```

8.1.5. The Service

Following the operation concept for services, as given in Section 5.4, the last remaining step for the service developer to derive functional services is to build docker images that can be run, e.g. on Kubernetes. It is necessary to build two distinct images, one for the **API** (which includes the data model) and one for the worker (which includes the data model and the forecasting or optimization code). The build instructions for both images are implemented as Dockerfile⁶⁹, see Listings 5 and 6.

Listing 5. *Dockerfile for the **API** container.*

```

FROM energy-service-generics:latest-service

COPY api.py data_model.py worker.py /source/service/

CMD ["/source/service/api.py"]

```

Listing 6. *Dockerfile for the worker container.*

```

FROM energy-service-generics:latest-service-pandas

RUN pip install pvlib scipy
COPY data_model.py fooc.py worker.py /source/service/

ENTRYPOINT [ "celery" ]
CMD [ "--app", "worker", "worker", "--loglevel=INFO" ]

```

8.2. Client Implementation

Above, see **NFR10**, we have argued that a key requirement for the widespread integration of services into **EMSs** is the ease of client implementation. In order to demonstrate the latter we provide an example for a minimal client, implemented as shell script, capable of retrieving **PV** power generation forecasts from the service introduced in the previous section in Listing 7. It should be noted that the example assumes that the service is reachable via the **localhost** network address, i.e. that the client is executed on the same machine as the service. Note further that the latest version of the code listed below, i.e. the version adapted to potential future changes of the framework, is provided as part of the **ESG** repository⁷⁰.

Listing 7. *Minimal example of a client script that allows retrieving **PV** power generation forecasts from the example service.*

⁶⁹<https://docs.docker.com/reference/dockerfile/>

⁷⁰https://github.com/fzi-forschungszentrum-informatik/energy-service-generics/tree/main/docs/examples/minimial_client


```

SERVICE_BASE_URL=${SERVICE_BASE_URL:-http://localhost:8800}
SERVICE_VERSION="test_version"

# Request a PV power generation forecast from the basic example service.
response=$(
  curl -X 'POST' \
    "${SERVICE_BASE_URL}/${SERVICE_VERSION}/request/" \
    -d '{
      "arguments": {
        "geographic_position": {
          "latitude": 49.01365,
          "longitude": 8.40444
        }
      },
      "parameters": {
        "pv_system": {
          "azimuth_angle": 0,
          "inclination_angle": 30,
          "nominal_power": 15
        }
      }
    }'
)

# Extract the ID of the task from the JSON response.
task_ID=$(echo $response | jq -r .task_ID)

# Poll status endpoint until status is ready.
status="unknown"
while [ $status != "ready" ]
do
  sleep 1
  response=$(
    curl -X 'GET' \
      "${SERVICE_BASE_URL}/${SERVICE_VERSION}/request/${task_ID}/status/"
  )
  status=$(echo $response | jq -r .status_text)
done

# Fetch and print the result.
response=$(
  curl -X 'GET' \
    "${SERVICE_BASE_URL}/${SERVICE_VERSION}/request/${task_ID}/result/"
)
echo $response | jq

```

One can perceive from inspecting the code above that the client logic is indeed very simple, thus easy to implement. The script is written in standard Unix Shell syntax and uses only two additional packages, `curl` for making [HTTP](#) requests and `jq` for parsing [JSON](#). The script follows the [API](#) concept derived in [Section 5.1](#). That is, a request is created with the first `curl` call. The large nested structure underneath defines the input arguments and parameters that are provided to the service. More discussion about the latter, including an explanation about the fields, is provided in [Section 8.1.2](#). Next, the script extracts the task ID of the created request and polls the `/status/` endpoint until the task has reached the `ready` status and

finally fetches the result. Note that the interaction with the `/fit-parameters/` endpoint of the service follows the same pattern and is thus omitted here for brevity.

While the service integration used in a production EMS would likely require more functionality, e.g. to parse the result into the format the EMS expects or to handle network errors, the concept of the client always remains the same. In fact, the ESG package contains a ready-to-use generic client⁷¹ which provides this additional functionality and reduces the implementation effort further.

Listing 8. *Example of a client script that allows retrieving PV power generation forecasts from the example service using the generic client provided in the ESG package.*

```
import os

from esg.clients.service import GenericServiceClient
from esg.models.base import _BaseModel
from esg.models.datapoint import ValueMessageList
from esg.models.metadata import GeographicPosition, PVSystem
from esg.service.worker import compute_request_input_model
from pydantic import Field

SERVICE_BASE_URL = os.getenv("SERVICE_BASE_URL")

class RequestArguments(_BaseModel):
    geographic_position: GeographicPosition

class FittedParameters(_BaseModel):
    pv_system: PVSystem

RequestInput = compute_request_input_model(
    RequestArguments=RequestArguments, FittedParameters=FittedParameters
)

class RequestOutput(_BaseModel):
    power_prediction: ValueMessageList = Field(description="Prediction of power production in W")

client = GenericServiceClient(
    base_url=SERVICE_BASE_URL, InputModel=RequestInput, OutputModel=RequestOutput
)

client.post_obj(
    input_data_obj={
        "arguments": {"geographic_position": {"latitude": 49.01365, "longitude": 8.40444}},
        "parameters": {
            "pv_system": {"azimuth_angle": 0, "inclination_angle": 30, "nominal_power": 15}
        },
    }
)

print(client.get_results_obj())
```

The code in Listing 8 demonstrates that the generic client class, which is provided as part of the ESG package, reduces the effort to the implementation of the data models as well as configuring the endpoint of the service the client should use. It is worth noting that specifying

⁷¹<https://github.com/fzi-forschungszentrum-informatik/energy-service-generics/blob/main/source/esg/clients/service.py>

the data models has been left on purpose for the developer of the client, although it is technically possible that the client fetches the latter automatically from the service. However, the idea is that the implementation of the data model on the client side documents the data structure that the downstream application, i.e. the [EMS](#), is designed for. That is, it allows the client application to detect any changes in the format of the data provided by the service. While such changes should not happen in theory, see discussion in [NFR08](#) and about versioning in Section 5.1, they might still occur due to mistakes made by developers of services. Here, it is likely much simpler to debug an error that is thrown directly in the client code than some error deep downstream in the application using the data, which might have no obvious direct connection to the service and its data format.

Finally, we would like to point out that the generic client provided by the [ESG](#) package can only be used in applications capable of executing Python programs. Any developer working on an application not capable of the latter will likely need to implement the client logic from scratch. However, the effort for such an activity should be rather limited as the generic client has been implemented in less than 400 lines of code. As an alternative, the developer could resort to partly automatically generate the client program using Swagger Codegen, see Section 6.2 for further details.

8.3. Scalability of Services

One of the core claims of this work is that the presented framework enables operation of forecasting and optimization services for potentially thousands of [EMSs](#). In order to demonstrate that this claim is indeed legitimate, we present an experiment that assesses the scalability in the present section.

8.3.1. Experiment Design

The goal of the experiment design described here is to create a situation in which we can examine the influence of horizontal scaling on the operation of a service using the proposed framework. In particular, the experiment is intended to investigate the scalability related to communication overhead between worker and [API](#) instances. The latter is of particular importance as, given sufficient available funds, compute resources can be bought in nearly infinite amounts, which means that the actual limiting factor to scalability is the inter-process communication, i.e. in our case the communication between [API](#) and worker containers as well as the intermediary message broker.

In contrast, very limited computing resources have been available for this experiment. In particular, the experiment was carried out on a single virtualized server with access to 64GiB of main memory and 12 cores of an Intel Xeon 4116 [CPU](#). We used Ubuntu 22.04⁷² as operating system and Microk8s⁷³, a Kubernetes distribution especially suitable for research and development, as container orchestration engine. A dedicated service, henceforth referred to as *scalability tester service*, has been implemented using the [ESG](#) framework for the sake of this experiment. The implementation consists of the previously described components, i.e. data model, worker, [API](#), as well as forecasting or optimization code. Especial consideration has been devoted to designing the latter as, given the limited resources and desired high number of tasks, it is obviously not possible to conduct any operation which requires a non-neglectable amount of compute resources. On the other hand, the component representing the forecasting or optimization code should provoke a realistic call pattern by the clients. That is, the clients will usually poll the `/status/` endpoint several times before the result becomes available. We expect this polling mechanism to have a significant impact on the communication load inside a

⁷²<https://ubuntu.com/>

⁷³<https://microk8s.io/>

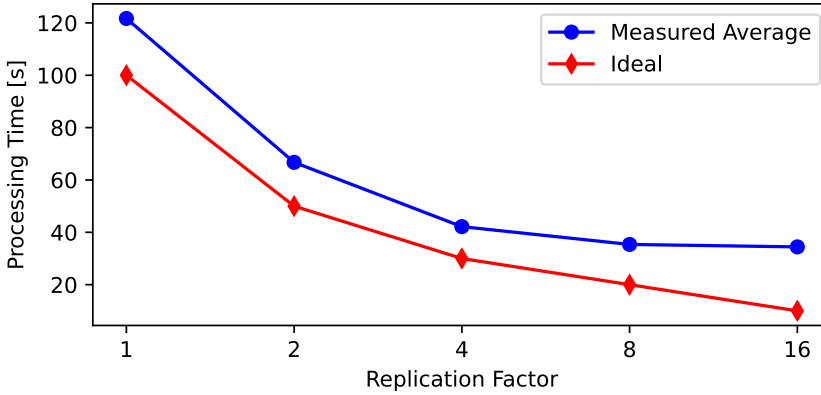


Figure 6. Measured and ideal time for processing 10.000 requests over replication factor.

service, as every status call triggers a lookup operation on the message broker. In order to satisfy both demands, we have implemented a simple sleep operation of ten seconds representing the forecasting or optimization code, as it does not induce CPU load while still blocking the worker and preventing immediately available results. Furthermore, a code simulating clients has been implemented. The latter uses the 100 instances⁷⁴ of the generic client (as introduced in Section 8.2) that issue in total 10.000 requests in very close time proximity. Utilizing the standard settings for the generic client, each client polls the `/status/` endpoint once per second and retrieves the results as they become ready. It is worth noting that each worker instance is configured to spawn 1000 threads, i.e. is capable of processing 1000 tasks in parallel. From this follows that there is an ideal bound for the processing time of the requests that is dependent on the number of worker instances. For example, neglecting all communication overhead, two workers of the scalability tester service are capable of processing 2.000 requests (because of the 1.000 threads per worker) every 10 seconds (due to the sleep time of 10 seconds representing the forecasting or optimization code), thus leading to a minimum compute time of 50 seconds for all 10.000 requests. Further details about the implementation of the scalability tester service and the corresponding clients are omitted here for brevity. However, the source code is provided in the ESG repository⁷⁵.

8.3.2. Experiment Execution and Results

In order to execute the experiment, the scalability tester service was deployed to the single-node Kubernetes instance introduced above. Utilizing the latter, the API and worker instances were scaled to a selected number, henceforth referred to as *replication factor*, i.e. it was taken care that this many instances of each the worker and API container were executed. The API and worker containers were using a single-node Redis⁷⁶ instance as message broker, a popular choice given the implementation of the ESG package⁷⁷. The actual experiment was conducted by executing the code invoking the clients. The latter was run on the personal laptop of one of the authors that was connected over the public internet to the service for extended realism.

⁷⁴It has not been possible to use a single client for each call as this exhausted the network resources, i.e. the number of dynamic ports, on the machine executing the respective script.

⁷⁵https://github.com/fzi-forschungszentrum-informatik/energy-service-generics/tree/main/docs/examples/scalability_example

⁷⁶<https://redis.io/>

⁷⁷<https://docs.celeryq.dev/en/stable/getting-started/backends-and-brokers/index.html#redis>

The result, as shown in Figure 6, is the average elapsed time starting directly before the program issued the first request and ending after all results have been available and transferred back. The measurements were repeated three times and the difference between the minimum and the maximum of the measured times were for all replication factors one second or less, the results appear thus robust and reliable. It can be seen that the time required for processing 10.000 requests shrinks from initially ca. 120 to approx. 35 seconds as the replication factor is increased from 1 over 2, 4, 8, and finally 16. It is thus clearly possible to mitigate a growing number of requests by replicating worker and [API](#) instances as the total number of requests per time unit is reciprocal to the processing time required for handling a fixed number of requests.

An interesting observation from the inspection of the measured results is that increasing the replication factor from 8 to 16 yields only a rather small gain of one second in real processing time, while the ideal processing times suggest a significant reduction from 20 to 10 seconds. Analysis of the [CPU](#) and memory consumption of the machine on which the experiment was executed on yielded no evidence that exhausted hardware could have been the reason. However, it appears likely that this behavior has been caused by the high number of threads (16.000 with the largest replication factor!), which very likely imposes a significant overhead for the operating system to switch tasks between these. In the latter case distributing the worker instances over more than one machine should improve the performance even further. While, at the time of writing, we do not possess the technical resources to validate this claim experimentally, we plan to catch up on this matter once we have access to a more sophisticated and better equipped Kubernetes cluster.

Finally, it is worth mentioning that the processing time of 35 seconds for 10.000 requests is roughly equivalent to 250.000 [EMSs](#) issuing one request per 15 minutes, which appears a solid foundation for forecasting and optimization services at scale. However, we estimate that using more sophisticated computing resources, it should rather easily be possible to serve several millions of [EMSs](#).

8.4. Comparison of Requirements with Concept and Implementation

As last part of our evaluation, this section presents a systematic comparison of the requirements derived in this work with the realization of our proposed framework and community concept. This comparison is provided in Table 2 for the functional and non-functional requirements defined in Section 4.2 and 4.3. One can easily perceive that our contributions, i.e. design and implementation of the framework as well as the community concept, do actually satisfy all requirements by inspecting the provided table.

As the requirements have been carefully derived from the current state of the art of energy management applications, it is concluded that our proposed framework and community concept are indeed valuable tools for the implementation and widespread distribution of forecasting and optimization services.

Table 2: Comparison of functional and non-functional requirements with actual realization in service framework and community concept.

ID	Description	Realization
FR01	A service developer must be able to derive a functional service with the service framework from existing forecasting or optimization code.	A service developer can derive a functional service by extending the generic components provided by the service framework with service specific components. See Section 5.2 for details.
FR02	An EMS must be able to interact with the service over a web API provided by the service.	An EMS can interact with a service (that utilizes the service framework) over a REST API. Further details can be found in Sections 5.1 (design) and 6.2 (implementation).
FR03	An EMS must be able to <i>request</i> a forecast or optimized schedule utilizing the API of the service.	An EMS can request a forecast or optimized schedule from a service (that utilizes the service framework) by interacting with the <code>/request/</code> endpoints defined in Section 5.1.
FR04	A service developer must be able to specify the format of the input and output data exchanged due to an EMS request for a forecast or optimized schedule from the service API.	A service developer can specify the format of the input and output data for the <code>/request/</code> endpoints by defining the data model. The concept is defined in Section 5.2, and a practical example is provided in Section 8.1.2.
FR05	An EMS must be able to fit system-specific parameters of a service utilizing its API.	An EMS can fit system-specific parameters of a service (that utilizes the service framework) by interacting with the <code>/fit-parameters/</code> endpoints defined in Section 5.1.
FR06	A service developer must be able to specify the format of the input and output data exchanged while an EMS interacts with the API of a service to fit the system-specific parameters.	A service developer can specify the format of the input and output data for the <code>/fit-parameters/</code> endpoints by defining the corresponding data model. The concept is defined in Section 5.2, and a practical example is provided in Section 8.1.2.
FR07	An EMS must have the option to store fitted system-specific parameters locally.	An EMS can fetch the system-specific parameters (i.e. the output of the interaction with the <code>/fit-parameters/</code> endpoints) after the service (that utilizes the service framework) has finished the fitting process, see Section 5.1. In fact, the service framework does not provide a functionality that would allow services to permanently store the fitted parameters. Further details are provided in Appendix A.

FR08	An EMS must be able to make calls to the API of the service which may take several hours to compute.	An EMS can make calls that take several hours (or much longer) to compute to the <code>/request/</code> and <code>/fit-parameters/</code> endpoints of a service (that utilizes the service framework) by first posting the demand for computation, then calling the respective <code>/status/</code> endpoint and finally fetching the result from the respective <code>/result/</code> endpoint, see Section 5.1 for details.
FR09	A service developer should be able to automatically generate a documentation for the API of a service.	A service developer can automatically generate an interactive documentation for the API of a service-based on the data model. See Section 6.2 for details.
NFR01	A service provider must be able to operate services with high availability and scalability.	A service provider can operate services (that utilize the service framework) with high availability and scalability as the operation concept (see Section 5.4) for services explicitly considers execution on clusters. Furthermore, the internal architecture of the services (see Section 5.3) supports scaling of API and worker processes over multiple machines, an important preliminary for high availability and performance on clusters.
NFR02	An EMS must be able to communicate with the service over an encrypted connection.	An EMS can communicate with services over an encrypted connection as the operation concept (see Section 5.4) demands that the service provider operates a gateway application in front of any service. The gateway encrypts the communication between client and service with the HTTPS protocol.
NFR03	A service provider must be able to restrict access to a service to authorized EMSs .	A service provider can restrict access to a service (that utilizes the service framework) by configuring the latter to verify that incoming calls contain a valid JWT . A general discussion of this concept can be found in Section 5.1 , while further implementation details and configuration options are provided in the online documentation of the service framework.
NFR04	A service developer/provider should be able to validate the correct implementation of the service framework.	A service developer/provider can validate the correct implementation of the service framework as it is published as open-source repository, see Section 7 for details.
NFR05	A service developer/provider should be able to verify that the service framework is actively maintained.	A service developer/provider can verify that the service framework is actively maintained by checking the website of the Open Energy Services community or directly contacting members of the latter, see Section 7 for details.

NFR06	A service developer should be able to derive a service with minimal effort from an existing forecasting or optimization algorithm.	Utilizing the service framework, a service developer can derive a functional service with minimal effort as all functionality generic to services in general is concentrated in the implementation of the service framework, which is readily provided. See Section 5.2 for conceptual details as well as Section 8.1 for an example demonstrating the implementation of a simple PV power forecast service.
NFR07	A service provider should be able to operate a service without requiring expert knowledge about IT infrastructure.	A service provider can operate a service without requiring expert knowledge about IT infrastructure as this is explicitly considered in the operation concept (see Section 5.4). Concrete measures include support for Docker Swarm as orchestration system as well as the possibility to use an IdP of an allied service provider.
NFR08	An EMS developer must be able to specify which version of a service should be utilized.	An EMS developer can specify which version of a service should be utilized by filling in the desired version in the {version} placeholder that is a part of all API endpoints, see Section 5.1 for details.
NFR09	An EMS developer should be able to quickly understand the API of a service by utilizing the documentation.	An EMS developer can inspect the automatically generated interactive documentation, the gold standard regarding comprehensibility, of any service (that utilizes the service framework). See Section 6.2 for details.
NFR10	An EMS developer should be able to implement a client to interact with the API of a service with minimal effort.	An EMS developer can use the generic service client provided as part of the ESG framework, see Section 8.2 for a practical example. As an alternative, it is possible to automatically generate the API-related code of a client using the freely available Swagger Codegen tool for any service (that utilizes the service framework). See Section 6.2 for details.

9. Conclusion and Outlook

The aim of this paper is to support the widespread adoption of EMSs in order to unlock flexibility and energy savings potentials of end consumers. We claim that economic viability is a severe issue for the utilization of EMSs at scale and that the provisioning of forecasting and optimization algorithms as a service can make a major contribution to achieving it. To this end, we introduce a software framework that allows the derivation of fully functional services from existing forecasting or optimization code with ease. Our development of this framework is strictly systematic and begins by deducing requirements from an extensive analysis of the application of EMSs in several domains. Based on this, we derive a holistic design concept for the framework, covering the components, the architecture, and the operation of services. We derive the ESG package from the proposed design concept, and we publish it as free and open-source software alongside this work. Beyond the service framework, this paper furthermore marks the starting point of the *Open Energy Service* community, our effort to continuously maintain the service framework but also provide ready-to-use forecasting and optimization services, to bootstrap future research projects and to accelerate the widespread adoption of

EMSs. This community is open for others to join, and any interest in participating is appreciated. Finally, we demonstrate that our framework and our community concept are valuable contributions that meet the goals of this work. To this end, we provide practical examples for the implementation of a service based on a simple **PV** power generation forecasting code, as well as the corresponding client. Furthermore, we demonstrate that our framework is capable of supporting the operation of services at relevant scales for **EMS** applications. We thus conclude that this work is a relevant step forward towards unlocking the potentials of forecasting and optimization algorithms provided as services for **EMSs**, which hopefully supports the utilization of energy management applications at scale.

Acknowledgments. We would like to thank the anonymous reviewers for their constructive feedback, which helped us to improve this paper. We furthermore like to thank Antonia Dieterich, who supported our analysis of related work.

Funding Statement. This research has partly been funded by the German Federal Ministry for Economic Affairs and Climate Action within the projects FlexBlue and AMAZING, the Helmholtz Association under the Program Energy System Design as well as the European Union within the project WeForming.

Supported by:



Federal Ministry
for Economic Affairs
and Climate Action

on the basis of a decision
by the German Bundestag



Co-funded by
the European Union

Competing Interests. None.

Data Availability Statement. None.

Ethical Standards. The research meets all ethical guidelines, including adherence to the legal requirements of the study country.

Author Contributions. Conceptualization: D.W; K.F; R.M. Funding acquisition: D.W. T.R; V.H; H.S. Methodology: D.W; L.L. Project administration: D.W. Software: D.W. Supervision: R.M; V.H; H.S; Validation: D.W. Writing – original draft: D.W; K.F; T.R; N.F. Writing – review & editing: D.W; K.F; T.R; N.F; R.M; V.H; H.S. All authors approved the final submitted draft.

Supplementary Material. The *Energy Service Generics* repository, including the source code as well as extensive documentation, can be found at:

<https://github.com/fzi-forschungszentrum-informatik/energy-service-generics>

Further details and latest news about the *Open Energy Services* community are provided on the corresponding website:

<https://open-energy-services.org/>

List of Abbreviations

API Application Programming Interface.

BAS Building Automation System.

BSS Battery Storage System.

CPU Central Processing Unit.

EMS Energy Management System.

ESG Energy Service Generics.

GUI Graphical User Interface.

HAL Hardware Abstraction Layer.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

HVAC Heating, Ventilation and Air Conditioning.

IdP Identity Provider.

JSON JavaScript Object Notation.

JWT JSON Web Token.

ML Machine Learning.

OIDC OpenID Connect.

PV Photovoltaic.

RBC Rule Based Control.

REST Representational State Transfer.

RPC Remote Procedure Call.

URL Uniform Resource Locator.

References

- [Al-Ghaili et al., 2021] Al-Ghaili, A. M., Kasim, H., Al-Hada, N. M., Jørgensen, B. N., Othman, M., and Wang, J. (2021). Energy Management Systems and Strategies in Buildings Sector: A Scoping Review. *IEEE Access*, 9:63790–63813.
- [Alebrahim et al., 2014] Alebrahim, A., Heisel, M., and Meis, R. (2014). A Structured Approach for Eliciting, Modeling, and Using Quality-Related Domain Knowledge. In *Computational Science and Its Applications – ICCSA 2014*, pages 370–386, Cham. Springer International Publishing.
- [Alizadeh et al., 2016] Alizadeh, M., Parsa Moghaddam, M., Amjady, N., Siano, P., and Sheikh-El-Eslami, M. (2016). Flexibility in future power systems with high renewable penetration: A review. *Renewable and Sustainable Energy Reviews*, 57:1186–1193.
- [Alves et al., 2006] Alves, C. F., Pereira, S., and de Castro, J. B. (2006). A study in market-driven requirements engineering. In *WER*, pages 61–66.
- [Anand et al., 2023] Anand, H., Nateghi, R., and Alemazkoo, N. (2023). Bottom-up forecasting: Applications and limitations in load forecasting using smart-meter data. *Data-Centric Engineering*, 4:e14.
- [Anthi et al., 2018] Anthi, E., Javed, A., Rana, O., and Theodorakopoulos, G. (2018). Secure Data Sharing and Analysis in Cloud-Based Energy Management Systems. In *Cloud Infrastructures, Services, and IoT Systems for Smart Cities*, pages 228–242, Cham. Springer International Publishing.
- [Antonelli et al., 2012] Antonelli, L., Rossi, G., do Prado Leite, J. C. S., and Oliveros, A. (2012). Deriving requirements specifications from the application domain language captured by Language Extended Lexicon. In *Proc. Workshop in Requirements Engineering (WER)*, pages 24–27.
- [Berkes and Keshav, 2024] Berkes, A. and Keshav, S. (2024). SOPEVS: Sizing and Operation of PV-EV-Integrated Modern Homes. In *Proceedings of the 15th ACM International Conference on Future and Sustainable Energy Systems*, e-Energy '24, page 14–26, New York, NY, USA. Association for Computing Machinery.
- [Blechmann et al., 2023] Blechmann, S., Sowa, I., Schraven, M. H., Streblow, R., Müller, D., and Monti, A. (2023). Open source platform application for smart building and smart grid controls. *Automation in Construction*, 145:104622.
- [Boiko et al., 2024] Boiko, O., Komin, A., Malekian, R., and Davidsson, P. (2024). Edge-Cloud Architectures for Hybrid Energy Management Systems: A Comprehensive Review. *IEEE Sensors Journal*, 24(10):15748–15772.
- [Chawda et al., 2020] Chawda, G. S., Shaik, A. G., Shaik, M., Padmanaban, S., Holm-Nielsen, J. B., Mahela, O. P., and Kaliannan, P. (2020). Comprehensive Review on Detection and Classification of Power Quality Disturbances in Utility Grid With Renewable Energy Penetration. *IEEE Access*, 8:146807–146830.

- [Chen et al., 2019] Chen, B., Cai, Z., and Bergés, M. (2019). Gnu-RL: A Precocial Reinforcement Learning Solution for Building HVAC Control Using a Differentiable MPC Policy. In *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, BuildSys '19, pages 316–325, New York, NY, USA. Association for Computing Machinery.
- [Cirillo et al., 2019] Cirillo, F., Solmaz, G., Berz, E. L., Bauer, M., Cheng, B., and Kovacs, E. (2019). A Standard-Based Open Source IoT Platform: FIWARE. *IEEE Internet of Things Magazine*, 2(3):12–18.
- [Coelho and Valente, 2017] Coelho, J. and Valente, M. T. (2017). Why Modern Open Source Projects Fail. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017*, pages 186–196, New York, NY, USA. Association for Computing Machinery.
- [Dawson-Haggerty et al., 2013] Dawson-Haggerty, S., Krioukov, A., Taneja, J., Karandikar, S., Fierro, G., Kitaev, N., and Culler, D. (2013). BOSS: Building Operating System Services. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 443–457, Lombard, IL. USENIX Association.
- [De Jongh et al., 2022] De Jongh, S., Gielnik, F., Mueller, F., Schmit, L., Suriyah, M., and Leibfried, T. (2022). Physics-informed geometric deep learning for inference tasks in power systems. *Electric Power Systems Research*, 211:108362.
- [Dengler et al., 2023] Dengler, G., Lalbakhsh, P., Bazan, P., Dayaratne, T., Liebmann, A., and German, R. (2023). P4 Poster abstract: A flexible simulation-optimization framework for smart grids using distributed agents. In *Abstracts of the 12th DACH+ Conference on Energy Informatics 2023*. Springer.
- [Ding et al., 2019] Ding, X., Du, W., and Cerpa, A. (2019). OCTOPUS: Deep Reinforcement Learning for Holistic Smart Building Control. In *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, BuildSys '19, pages 326–335, New York, NY, USA. Association for Computing Machinery.
- [Eckert et al., 2019] Eckert, R., Stuermer, M., and Myrach, T. (2019). Alone or Together? Inter-organizational Affiliations of Open Source Communities. *Journal of Systems and Software*, 149:250–262.
- [Eurostat, 2023] Eurostat (2023). Energy consumption in households. https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Energy_consumption_in_households.
- [Fielding, 2000] Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.
- [Fogel, 2022] Fogel, K. (2022). Producing Open Source Software. <https://producingoss.com/en/index.html>.
- [Förderer et al., 2022] Förderer, K., Hagenmeyer, V., and Schmeck, H. (2022). Automated Generation of Models for Demand Side Flexibility Using Machine Learning: An Overview. *SIGENERGY Energy Inform. Rev.*, 1(1):107–120.
- [Galenzowski et al., 2023] Galenzowski, J., Waczowicz, S., Meisenbacher, S., Mikut, R., and Hagenmeyer, V. (2023). A real-world district community platform as a cyber-physical-social infrastructure systems in the energy domain. In *Proceedings of the 10th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, pages 434–441.
- [Glinz, 2007] Glinz, M. (2007). On Non-Functional Requirements. In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 21–26, Delhi. IEEE.
- [Gwerder et al., 2013] Gwerder, M., Gyalistras, D., Sagerschnig, C., Smith, R. S., and Sturzenegger, D. (2013). Final Report: Use of Weather And Occupancy Forecasts For Optimal Building Climate Control Part II: Demonstration (OptiControl-II). Technical report, ETH Zürich.
- [Han et al., 2023] Han, B., Zahraoui, Y., Mubin, M., Mekhilef, S., Seyedmahmoudian, M., and Stojcevski, A. (2023). Home Energy Management Systems: A Review of the Concept, Architecture, and Scheduling Strategies. *IEEE Access*, 11:19999–20025.
- [Henggeler Antunes et al., 2022] Henggeler Antunes, C., Alves, M. J., and Soares, I. (2022). A comprehensive and modular set of appliance operation MILP models for demand response optimization. *Applied Energy*, 320:119142.
- [Hill et al., 2023] Hill, A., Pieper, C., Bruhn, J.-H., Schönfeldt, P., and Penaherrera Vaca, F. A. (2023). P2 Poster abstract: District energy management simulation framework with rolling horizon approach. In *Abstracts of the 12th DACH+ Conference on Energy Informatics 2023*. Springer.
- [Hofmeister et al., 2024] Hofmeister, M., Bai, J., Brownbridge, G., Mosbach, S., Lee, K. F., Farazi, F., Hillman, M., Agarwal, M., Ganguly, S., Akroyd, J., and Kraft, M. (2024). Semantic agent framework for automated flood assessment using dynamic knowledge graphs. *Data-Centric Engineering*, 5:e14.
- [Hunter, 2017] Hunter, K. L. (2017). *Irresistible APIs: Designing Web APIs That Developers Will Love*. Manning, Shelter Island, NY.
- [IEEE, 2002] IEEE (2002). IEEE Standard Glossary of Software Engineering Terminology.
- [IPCC, 2022] IPCC (2022). Summary for Policymakers. In Shukla, P., Skea, J., Reisinger, A., Slade, R., Fradera, R., Pathak, M., Khouradajie, A. A., Belkacemi, M., van Diemen, R., Hasija, A., Lisboa, G., Luz, S., Malley, J., McCollum, D., Some, S., and Vyas, P., editors, *Climate Change 2022: Mitigation of Climate Change. Contribution of Working Group III to the Sixth Assessment Report of the Intergovernmental Panel*

- on *Climate Change*, pages 3–48. Cambridge University Press, 1 edition.
- [ISO/IEC, 2023] ISO/IEC (2023). *ISO/IEC 25010: Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model*. ISO copyright office.
- [Jawarneh et al., 2019] Jawarneh, I. M. A., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., Montanari, R., and Palopoli, A. (2019). Container Orchestration Engines: A Thorough Functional and Performance Comparison. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pages 1–6.
- [Jin et al., 2018] Jin, B., Sahni, S., and Shevat, A. (2018). *Designing Web APIs: Building APIs That Developers Love*. O'Reilly, Beijing Boston Farnham, first edition.
- [Jones et al., 2015] Jones, M. B., Bradley, J., and Sakimura, N. (2015). JSON web token (JWT). RFC 7519, RFC Editor.
- [Khalid, 2024] Khalid, M. (2024). Smart grids and renewable energy systems: Perspectives and grid integration challenges. *Energy Strategy Reviews*, 51:101299.
- [Kondziella and Bruckner, 2016] Kondziella, H. and Bruckner, T. (2016). Flexibility requirements of renewable energy based electricity systems – a review of research results and methodologies. *Renewable and Sustainable Energy Reviews*, 53:10–22.
- [Kornienko et al., 2021] Kornienko, D. V., Mishina, S. V., Shcherbatykh, S. V., and Melnikov, M. O. (2021). Principles of securing RESTful API web services developed with python frameworks. *Journal of Physics: Conference Series*, 2094(3):032016.
- [Kotilainen, 2019] Kotilainen, K. (2019). *Energy Prosumers' Role in the Sustainable Energy System*, pages 1–14. Springer International Publishing, Cham.
- [Krebs and Cruz Martinez, 2022] Krebs, B. and Cruz Martinez, J. (2022). Developing RESTful APIs with Python and Flask.
- [Kursawe et al., 2011] Kursawe, K., Danezis, G., and Kohlweiss, M. (2011). Privacy-Friendly Aggregation for the Smart-Grid. In *Privacy Enhancing Technologies: 11th International Symposium, PETS 2011, Waterloo, ON, Canada, July 27-29, 2011. Proceedings 11*, pages 175–191. Springer.
- [Lamprecht et al., 2020] Lamprecht, A.-L., Garcia, L., Kuzak, M., Martinez, C., Arcila, R., Martin Del Pico, E., Dominguez Del Angel, V., Van De Sandt, S., Ison, J., Martinez, P. A., McQuilton, P., Valencia, A., Harrow, J., Psomopoulos, F., Gelpi, J. L., Chue Hong, N., Goble, C., and Capella-Gutierrez, S. (2020). Towards FAIR principles for research software. *Data Science*, 3(1):37–59.
- [Langer et al., 2013] Langer, L., Skopik, F., Kienesberger, G., and Li, Q. (2013). Privacy issues of smart e-mobility. In *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*, pages 6682–6687.
- [Lee et al., 2016] Lee, E.-K., Shi, W., Gadh, R., and Kim, W. (2016). Design and Implementation of a Microgrid Energy Management System. *Sustainability*, 8(11):1143.
- [Lenk et al., 2020] Lenk, S., Arnoldt, A., Rösch, D., and Bretschneider, P. (2020). Hardware/software architecture to investigate resilience in energy management for smart grids. In *2020 IEEE PES Innovative Smart Grid Technologies Europe (ISGT-Europe)*, pages 51–55. IEEE.
- [Loucopoulos and Champion, 1988] Loucopoulos, P. and Champion, R. (1988). Knowledge-based approach to requirements engineering using method and domain knowledge. *Knowledge-Based Systems*, 1(3):179–187.
- [Malviya and Dwivedi, 2022] Malviya, A. and Dwivedi, R. K. (2022). A Comparative Analysis of Container Orchestration Tools in Cloud Computing. In *2022 9th International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 698–703.
- [Maree and Bagle, 2022] Maree, J. P. and Bagle, M. (2022). A Building Automation and Control micro-service architecture using Physics Inspired Neural Networks. *E3S Web of Conferences*, 362:13001.
- [Marinakakis et al., 2020] Marinakakis, V., Doukas, H., Tsapelas, J., Mouzakitis, S., Sicilia, Á., Madrazo, L., and Sgouridis, S. (2020). From big data to smart energy services: An application for intelligent energy management. *Future Generation Computer Systems*, 110:572–586.
- [Mateos-Garcia and Steinmueller, 2008] Mateos-Garcia, J. and Steinmueller, W. E. (2008). The institutions of open source software: Examining the Debian community. *Information Economics and Policy*, 20(4):333–344.
- [Mauser et al., 2015] Mauser, I., Hirsch, C., Kochannek, S., and Schmeck, H. (2015). Organic Architecture for Energy Management and Smart Grids. In *2015 IEEE International Conference on Autonomic Computing*, pages 101–108, Grenoble, France. IEEE.
- [Meisenbacher et al., 2023] Meisenbacher, S., Heidrich, B., Martin, T., Mikut, R., and Hagenmeyer, V. (2023). AutoPV: Automated photovoltaic forecasts with limited information using an ensemble of pre-trained models. In *Proceedings of the 14th ACM International Conference on Future Energy Systems*, pages 386–414, Orlando FL USA. ACM.
- [Mercl and Pavlik, 2019] Mercl, L. and Pavlik, J. (2019). The Comparison of Container Orchestrators. In Yang, X.-S., Sherratt, S., Dey, N., and Joshi, A., editors, *Third International Congress on Information and Communication Technology*, volume 797, pages 677–685. Springer Singapore, Singapore.
- [Mohamed et al., 2018] Mohamed, N., Al-Jaroodi, J., and Jawhar, I. (2018). Service-Oriented Big Data Analytics for Improving Buildings Energy Management in Smart Cities. In *2018 14th International Wireless*

- Communications & Mobile Computing Conference (IWCMC)*, pages 1243–1248, Limassol. IEEE.
- [Oldewurtel et al., 2012] Oldewurtel, F., Parisio, A., Jones, C. N., Gyalistras, D., Gwerder, M., Stauch, V., Lehmann, B., and Morari, M. (2012). Use of model predictive control and weather forecasts for energy efficient building climate control. *Energy and Buildings*, 45:15–27.
- [Papaeftymiou and Dragoon, 2016] Papaeftymiou, G. and Dragoon, K. (2016). Towards 100% renewable energy systems: Uncapping power system flexibility. *Energy Policy*, 92:69–82.
- [Pipattanasomporn et al., 2015] Pipattanasomporn, M., Kuzlu, M., Khamphanchai, W., Saha, A., Rathinavel, K., and Rahman, S. (2015). BEMOSS: An Agent Platform to Facilitate Grid-Interactive Building Operation with IoT Devices. In *2015 IEEE Innovative Smart Grid Technologies - Asia (ISGT ASIA)*, pages 1–6, Bangkok, Thailand. IEEE.
- [Pohl, 1996] Pohl, K. (1996). Requirements engineering: An overview. In *Encyclopedia of Computer Science and Technology*, volume 36 - supp. 21. CRC Press.
- [Regnell and Brinkkemper, 2005] Regnell, B. and Brinkkemper, S. (2005). Market-Driven Requirements Engineering for Software Products. In Aurum, A. and Wohlin, C., editors, *Engineering and Managing Software Requirements*, pages 287–308. Springer-Verlag, Berlin/Heidelberg.
- [Roccotelli et al., 2022] Roccotelli, M., Mangini, A. M., and Fanti, M. P. (2022). Smart District Energy Management With Cooperative Microgrids. *IEEE Access*, 10:36311–36326.
- [Rodriguez et al., 2018] Rodriguez, M. A., Cuenca, L., and Ortiz, A. (2018). FIWARE Open Source Standard Platform in Smart Farming - A Review. In Camarinha-Matos, L. M., Afsarmanesh, H., and Rezgui, Y., editors, *Collaborative Networks of Cognitive Systems*, pages 581–589, Cham. Springer International Publishing.
- [Ruhnau et al., 2019] Ruhnau, O., Bannik, S., Otten, S., Praktiknjo, A., and Robinius, M. (2019). Direct or indirect electrification? A review of heat generation and road transport decarbonisation scenarios for Germany 2050. *Energy*, 166:989–999.
- [Saeed and Abdallah, 2022] Saeed, L. and Abdallah, G. (2022). *Security with JWT*, pages 293–308. Apress, Berkeley, CA.
- [Salpakari and Lund, 2016] Salpakari, J. and Lund, P. (2016). Optimal and rule-based control strategies for energy flexibility in buildings with PV. *Applied Energy*, 161:425–436.
- [Schibuola et al., 2015] Schibuola, L., Scarpa, M., and Tambani, C. (2015). Demand response management by means of heat pumps controlled via real time pricing. *Energy and Buildings*, 90:15–28.
- [Scott and Neray, 2021] Scott, S. and Neray, G. (2021). Best practices for REST API security: Authentication and authorization. <https://stackoverflow.blog/2021/10/06/best-practices-for-authentication-and-authorization-for-rest-apis/>.
- [Shaikh et al., 2014] Shaikh, P. H., Nor, N. B. M., Nallagownden, P., Elamvazuthi, I., and Ibrahim, T. (2014). A review on optimized control systems for building energy and comfort management of smart sustainable buildings. *Renewable and Sustainable Energy Reviews*, 34:409–429.
- [Singh, 2023] Singh, R. (2023). Flask vs FastAPI: Which Python Web Framework is Right for You? <https://www.linkedin.com/pulse/flask-vs-fastapi-which-python-web-framework-right-you-ritwik-singh-ddxtc/>.
- [Sovacool et al., 2022] Sovacool, B. K., Barnacle, M. L., Smith, A., and Brisbois, M. C. (2022). Towards improved solar energy justice: Exploring the complex inequities of household adoption of photovoltaic panels. *Energy Policy*, 164:112868.
- [Späth, 2023] Späth, P. (2023). *REST Security*, pages 175–194. Apress, Berkeley, CA.
- [Srilakshmi and Singh, 2022] Srilakshmi, E. and Singh, S. P. (2022). Energy regulation of EV using MILP for optimal operation of incentive based prosumer microgrid with uncertainty modelling. *International Journal of Electrical Power & Energy Systems*, 134:107353.
- [Varenhorst et al., 2024] Varenhorst, I. A. M., Hoogsteen, G., Gerards, M. E. T., and Hurink, J. L. (2024). Enhancing Privacy Through Time Aggregation of Load Profiles in Energy Management. In *2024 IEEE 8th Energy Conference (ENERGYCON)*, pages 1–6.
- [Volk et al., 2017] Volk, K., Lakenbrink, C., Kurka, C., and Rupp, L. (2017). Grid-Control - An Overall Concept for the Distribution Grid of the "Energiewende". In *International ETG Congress 2017*, pages 1–6.
- [Washizaki, 2024] Washizaki, H., editor (2024). *Guide to the Software Engineering Body of Knowledge (SWEBOK): Version 4.0*. IEEE Computer Society Press, Washington, DC, USA, 4rd edition.
- [Wölffe et al., 2020] Wölffe, D., Vishwanath, A., and Schmeck, H. (2020). A Guide for the Design of Benchmark Environments for Building Energy Optimization. In *Proceedings of the 7th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, BuildSys '20, pages 220–229, New York, NY, USA. Association for Computing Machinery.
- [Xuereb Conti et al., 2023] Xuereb Conti, Z., Choudhary, R., and Magri, L. (2023). A physics-based domain adaptation framework for modeling and forecasting building energy systems. *Data-Centric Engineering*, 4:e10.
- [Zafar et al., 2020] Zafar, U., Bayhan, S., and Sanfilippo, A. (2020). Home Energy Management System Concepts, Configurations, and Technologies for the Smart Grid. *IEEE Access*, 8:119271–119286.

A. Interaction Patterns Between EMS and Service

In Section 4.2 we have discussed that it may be necessary to fit system-specific parameters to utilize the full potential of a forecasting or optimization service. Furthermore, we have pointed out that this fitting process will usually need some form of historic measurements and that some EMS users will want to keep these historic measurements on-premise while others will not be capable of doing so. In this section, we will contrast the resulting interaction patterns between EMS and service based on this important difference. To this end, we will return to the example used in the functional requirements section, i.e. a service that provides forecasts of PV power generation.

A.1. Local Storage of Measurements and Parameters

Let us first inspect the case of an EMS that keeps historic measurements on-premise, i.e. a system with an architecture like the one displayed in Figure 2. The interaction of such an EMS with the considered PV power generation forecast service would contain the following steps:

1. The EMS calls the API endpoint of the service related to the fitting process. This includes pushing historic measurements of the power generated by the PV system for which forecasts should be computed. Note that the EMS may need to repeat this step in a certain interval, especially once the forecasting performance deteriorates. Finding the right interval for refitting the parameters is solely the responsibility of the EMS, as the service provider has no access to the measured data and can thus not evaluate the performance of the forecasts provided by the service.
2. The EMS retrieves the system-specific parameters once the service has finished the fitting process. The EMS must store these parameters locally.
3. The EMS requests a forecast from the service by issuing a call to the respective API endpoint of the service (this is likely repeated with arbitrary periodicity). The call must contain the arguments (e.g. the coordinates of the target system) as well as the fitted parameters (as returned in the previous step).
4. The EMS retrieves the computed forecast once the service has finished computing it.

With this interaction pattern, the service cannot request any data of the HAL, as the EMS needs to actively push the data to the service. This means that the EMS (developer) is always in control over the data stored by the system.

A.2. Cloud Storage of Measurements and Parameters

The second case is an EMS that has been configured to push the relevant historic measurements to a data platform that is accessible to the service provider, as summarized in Figure 7

This scenario is closer to a conventional "as a service" construct and allows the service provider to take over responsibility and complexity from the EMS. The price for this convenience is that the historic measurements of the EMS are shared with the service provider, which might occur as a privacy issue for some users. The resulting interaction pattern between the EMS, data platform, and service contains the following steps:

1. The EMS continuously pushes all relevant measurements to the data platform. Furthermore, the EMS must upload all other information required to invoke the desired services (e.g. the coordinates representing the global position of the target PV system) to the data platform.

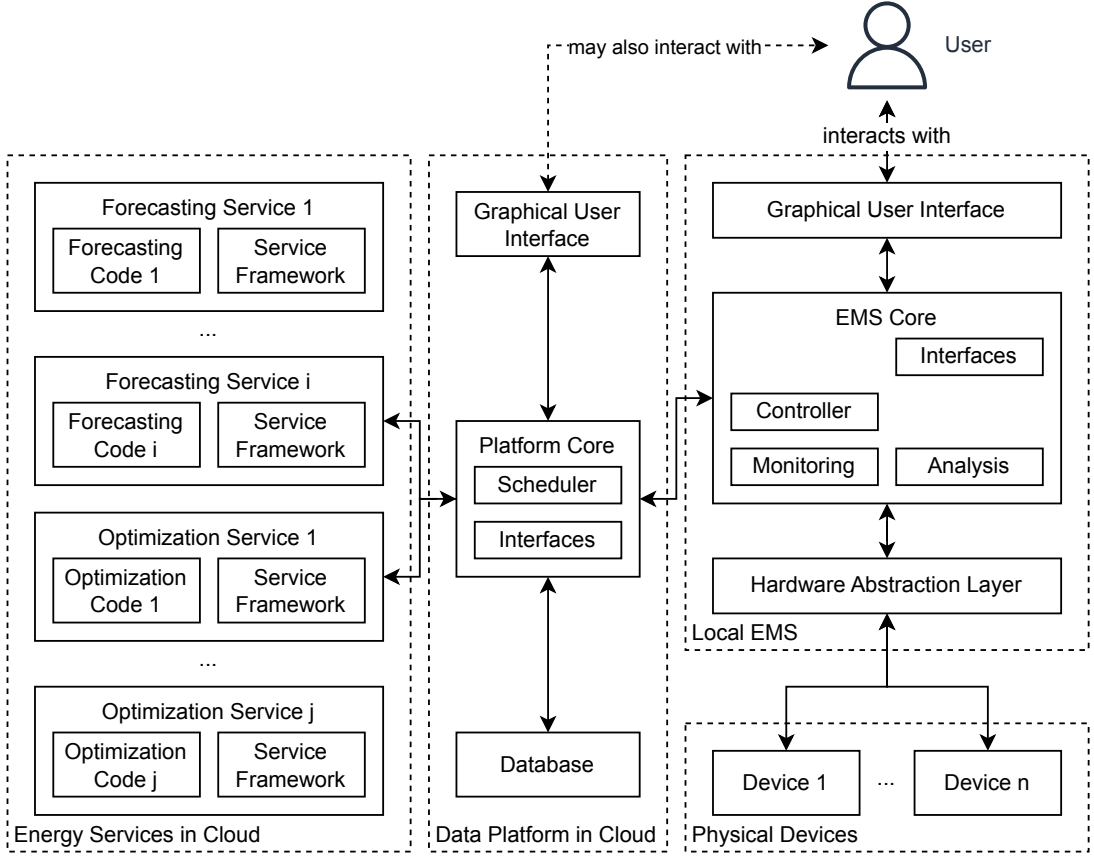


Figure 7. High-level architecture of an *EMS* indirectly utilizing forecasting and optimization services via a platform. Note that some functionality like Monitoring or Analysis may, in fact, be implemented as part of the cloud data platform instead.

2. The service provider⁷⁸ retrieves the relevant historic measurements from the data platform and uses these to call the *API* endpoint of the service related to the fitting process.
3. The service provider retrieves the system-specific parameters after the service has finished computing these and stores the parameters in the data platform⁷⁹.
4. At periodic intervals (e.g. every 15 minutes) or certain events (new information is available), the service provider retrieves the system-specific parameters as well as all other information to invoke the intended service and utilizes this data to request the forecast required by the *EMS* from the service.
5. Once the service has finished computing the forecast, the service provider collects it from the service and writes the forecast to the data platform.
6. The *EMS* periodically polls the data platform for new forecasts and retrieves these once available.

⁷⁸In fact this could be a third party that is not a service provider but invokes the services on behalf of the *EMS*, e.g. as a commercial offering. However, this does not change the interaction pattern and is thus neglected here.

⁷⁹Actually, it is not important whether the parameters are stored in the same data platform or some other database.