applied sciences

MDPI

Article

Architectural Concept and Evaluation of a Framework for the Efficient Automation of Computational Scientific Workflows: An Energy Systems Analysis Example

Jianlei Liu [1,*], Eric Braun [1,*], Clemens Düpmeier [1], Patrick Kuckertz [2], D. Severin Ryberg [2], Martin Robinius [2], Detlef Stolten [2,3] and Veit Hagenmeyer [1]

1 Institute for Automation and Applied Informatics (IAI), Karlsruhe Institute of Technology (KIT), Hermann-von-Helmholtz-Platz 1, 76344 Eggenstein-Leopoldshafen, Germany; clemens.duepmeier@kit.edu (C.D.); veit.hagenmeyer@kit.edu (V.H.)

2 Institute of Electrochemical Process Engineering (IEK-3), Forschungszentrum Jülich GmbH, Wilhelm-Johnen-Straße, 52428 Jülich, Germany; p.kuckertz@fz-juelich.de (P.K.); s.ryberg@fz-juelich.de (D.S.R.); m.robinius@fz-juelich.de (M.R.); d.stolten@fz-juelich.de (D.S.)

3 Chair of Fuel Cells, RWTH Aachen University, c/o Institute of Electrochemical Process Engineering (IEK-3), Forschungszentrum Jülich GmbH, Wilhelm-Johnen-Straße, 52428 Jülich, Germany

* Correspondence: jianlei.liu@kit.edu (J.L.); eric.braun2@kit.edu (E.B.)

Received: 14 January 2019; Accepted: 18 February 2019; Published: 20 February 2019

Abstract: Scientists and engineers involved in the design of complex system solutions use computational workflows for their evaluations. Along with growing system complexity, the complexity of these workflows also increases. Without integration tools, scientists and engineers are often highly concerned with how to integrate software tools and model sets, which hinders their original research or engineering aims. Therefore, a new framework for streamlining the creation and usage of automated computational workflows is introduced in the present article. It uses state-of-the-art technologies for automation (e.g., container-automation) and coordination (e.g., distributed message oriented middleware), and a microservice-based architecture for novel distributed process execution and coordination. It also supports co-simulations as part of larger workflows including additional auxiliary computational tasks, e.g., forecasting or data transformation. Using Apache NiFi, an easy-to-use web interface is provided to create, run and control workflows without the need to be concerned with the underlying computing infrastructure. Initial framework testing via the implementation of a real-world workflow underpins promising performance in the realms of parallelizability, low overheads and reliable coordination.

Keywords: automation; microservices; computing cluster; energy system; data processing; co-simulation; parallelization; coordination

1. Introduction

Traditional science often follows the path of formulating a problem statement, creating a scientific experiment as a verification environment and then running the experiment to gather data, that is then analyzed to either verify or falsify made assumptions. However, nowadays setting up real world environments for doing experimental evaluations of problem statements is often too costly and time consuming. Due to this, the experimental environment is often replaced by digital solutions using software models run in simulators and other scientific software tools for performing tasks, substituting the real world experimental workflow for gathering data with a digital workflow in silico (in the following called "computational scientific workflow"). The research field of computational science tackles challenges like these by creating necessary tools and research methodologies for offering software solutions supporting digital workflows performed by scientists of any research fields that need such software solutions. Aspects relevant to the respective research are formalized and transformed into computer models, which provide efficient means to analyze large sets of data intensive scenarios and utilize advanced computing capabilities. Across disciplines, the model landscape comprises a wide variety of model types, each focusing on certain facets of a system with individual levels of detail and resolution and applying different methodologies for simulation, optimization, and statistical data processing. To gain a broader view on a system, it is typically necessary to use multiple models, simulators and other auxiliary tools (e.g., optimization tools), and combine their input and output data flows, forming a more complex computational scientific workflow for the evaluation of key system properties. Such a workflow can be formally described by defining the connection between model output and input streams and the coordinated execution of the model logic to implement sequential, iterative, or simultaneous execution logic. If more than one simulator is involved, other more complex coordination mechanisms, e.g., time step-based or event-based synchronization of simulator execution, may be needed for modelling co-simulation workflows. To achieve a higher degree of workflow automation, the model tasks as part of an automated workflow can be complemented by any other sort of auxiliary processing task, e.g., tasks performing data transformation, validation, visualization, etc.

Figure 1 depicts the role of software instrumented workflows within the context of computational science, which provides a digital equivalent to the traditional experiment setup. Most often in this case, scientific research begins with the formulation of a research question that can be more or less concretely stated. The benefits of researching this topic and the impact of potential findings are described. After the question has been stated, typically a profound investigation, e.g., based on literature research of the thematic background, is conducted, identifying the most advanced models, procedures and processes that might be helpful in answering the question. The found models, procedures and processes must then be compiled into the digital equivalent of the classical experiment setup so that the digital experiments can then be performed by executing the digital workflow. Depending on the availability of prerequisites found by this investigation, the subsequent research approach is determined. If the found state-of-the-art models and methods are already implemented in a form fitting to the investigator's software-infrastructure, they can directly be integrated and used. If this is not possible, new model implementations or the implementations of dedicated software for e.g., implementing a certain control algorithm, become necessary. After all models and auxiliary software tools have been implemented, the foreseen digital experiment workflow can be (manually) executed by inputting adequate test data and gathering result data until all respective scenarios have been sufficiently investigated. The final model results, which are typically already post-processed and visualized for human readability at the end of a workflow, are then put into the context of the research topic and the results of previous scientific work. The outcomes are carefully interpreted and conclusions drawn within the respective scientific community. In the best case, a satisfying answer to the originally stated research question can be found. It is typically the case that research findings lead to new research questions requiring extensions and adjustments to the previous workflow, which will be the beginning of the next iteration within the workflow development cycle.

For implementing the coordination and interaction between workflow tasks, several approaches can be employed. In a tight coupling approach, concrete task implementations directly refer to each other in their source codes. This approach is constrained to the use of suitable programming frameworks for implementing the interprocess or distributed communication logic and requires an intimate knowledge of the individual models' internal procedures (white box). The immediate dependencies between model functionalities lead to a low level of modularity hindering reuse of model code in other contexts and internal refactoring. The replacement of individual model components or versions involves high reprogramming, testing, and documentation efforts. Loose coupling on the other hand aims for an abstraction of model components where each component implements a more generic extrinsic interface e.g., defining input and output functionality, allowing flexible configuration for adapting the module to different applications contexts and encapsulating the internal model logic (black box).

Utilizing this modular approach to implement a workflow, modules can be used in different application contexts and only knowledge about their respective interfaces is needed. The replacement of components within a workflow implementing a compatible extrinsic interface does not require code changes in other tasks but only requires adjustments to the runtime configuration responsible for executing the task component. The more the respective tasks use common interface standards for e.g., describing the exchange of data or configuration information, the less adjustments are needed to integrate these components in a workflow. Although a loose coupling approach allows for a higher degree of freedom in combining programming languages and environments, the design of a stable extrinsic interface providing compatibility with standard exchange interfaces to other related software tools has to explicitly be considered during the construction of a workflow.

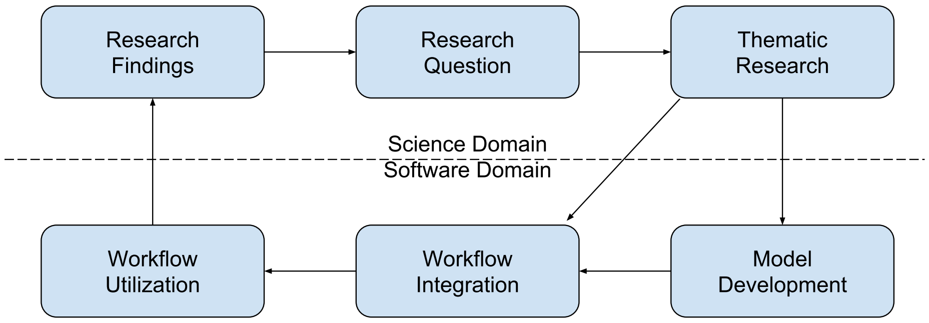


Figure 1. Development cycle of model workflows in the context of computational science.

In context of researching and evaluating the behavior of complex system setups, the manual executions of digital scientific workflows—where many manual steps are performed by using different software tools—become more and more tedious. Often, automated coordination of the execution of one or more software tools is needed to perform e.g., a co-simulation. Moreover, this form of coordination is programmed by scientists themselves-as an auxiliary tool by using one of the strategies introduced previously, or a certain framework (e.g., a co-simulation framework) is used, which performs the coordination but also requires writing glue code. This all contributes to a large overhead that scientists are confronted with when setting up their digital equivalent of an experimental setup, and that hinders them from concentrating on performing the actual experiments and evaluating the results. Thus, a central research question nowadays is to determine whether generic computational workflow execution platforms can be created to fully automate the execution of complex computational workflows and free the scientists from manually implementing different kinds of auxiliary tools for e.g., implementing coordination or data transformation.

Considering the fast growing complexity of modern technical systems and the need for efficiently performing many interconnecting software-based scientific tasks for their evaluation, an organization's development of auxiliary tools and setup of a software environment that facilitates easy-to-use workflows is a strategic issue with significant impact on research efficiency and long-term consequences. Against the background of the software used by the respective scientific community, the choices of programming languages and environments, operating systems, optimization solvers and other supporting tools and libraries, interface standards, file formats, etc. directly influence the potential to couple the organization's model set with those of fellow researchers. Since the investigation of complex and interdisciplinary research questions is usually not handled by a single organization, but is made possible through collaborations within the community, this potential is of particular importance. At the same time, changing an organization's software strategy is not an easy task. Models and other executables are already implemented, making redevelopments utilizing other programming languages and environments seldom worthwhile. Additionally, the retraining of employees and the rebuilding of programming experiences constitute cumbersome exercises. This makes coupling environments that allow the integration of existing tools and models with as little effort as possible very valuable.

In addition, while ensuring sufficient flexibility in model coupling, the requirements for efficiently processing the increasingly complex workflows—which is indispensable for the achievement of meaningful and reliable results—present a growing performance challenge. In contrast to white box integrated applications, which are often internally designed for parallel computing by using a parallel computing framework for implementation, loosely coupled workflows are usually constructed by chaining existing model components together without efficient parallel execution of the components in mind. It follows that the logic to run subprocesses and entire workflows in parallel as well as the communication control and consistency assurance mechanisms have to be designed, implemented, and tested with each newly drafted workflow. The workflow engineering and runtime environment should allow the instrumentation of such features as essential parts of a scientific computing workflow as easy as possible.

Within this overall context of efficiently performing computational science related tasks for providing such a scientific workflow environment as essential software infrastructure is an open research question, especially for energy research projects that bundle the smart energy systems related research of several research centers. One key element of such a research environment is a software platform that easily allows the research centers to not only share data but especially integrate their partly software solutions for performing digital workflows (e.g., models of new storage solutions, technical energy plants, optimization and control algorithms) into bigger, fully automated workflows that implement integrated energy system co-simulation models or other complex computational scientific workflows. The requirements that have to be met by such a platform can be summarized as follows:

- automated set up and execution of computational scientific workflows and co-simulations
- reusability of integrated executables with their specific dependencies and runtime environments
- configurable communication between executables without the need for changing executables or specific implementations of interfaces and adapters by the users
- availability of an easy-to-use web interface to build, operate and manage scenarios
- parallelization and coordination of workflows for increasing performance

The remainder of this article is organized as follows: Section 2 presents the main results of a literature review on already existing solutions for efficiently automating complex computational scientific workflows. Since no solution (either commercial or open source) was found that fulfills the discussed requirements, further research on developing a flexible framework as a solution for automating computational scientific workflows within the energy research software environment was conducted. Within this context, a background literature review was performed for finding state-of-the-art methodological approaches and reusable frameworks that could be used as building blocks for a comprehensive solution. The thereby selected technologies and used basic approaches are described in Section 2.2. In Section 3, the architectural design of the workflow automation framework is described. The architecture addresses the goals of a generic, modular, and highly scalable framework supporting the loose coupling of executables. Additionally, by instrumenting cluster computing environments and modern runtime automation technologies, it is capable of efficiently performing complex data processing and co-simulation workflows with high levels of performance and automation. Furthermore, a web-based editor and runtime interface is presented, providing an easy-to-use user interface for defining and managing scientific workflows. In Section 4, for the purpose of testing and evaluating the presented framework, a typical scientific model chain within the field of Energy Systems Analysis is used as an example workflow and implemented within the framework. The setup of this example workflow and the overall procedure for evaluating and benchmarking the framework are explained. The implications of the chosen architectural design and the benchmark results are then discussed in Section 5. Section 6 summarizes key results discussed in the article and gives an outlook on future work.

2. Related Work and State-Of-The-Art Technologies

In the following section, related works are discussed. Additionally, existing concepts and state-of-art technologies shown in Figure 2 used for implementation of the framework are presented.

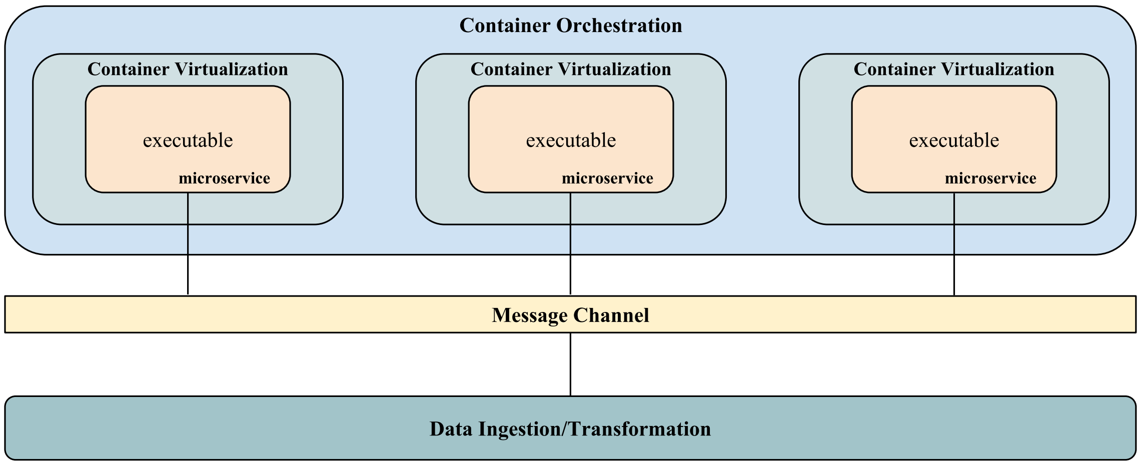


Figure 2. Concepts and state-of-the-art technologies used for the presented architecture.

2.1. Related Work

The Python programming language is nowadays often used by scientists to manually setup computational scientific workflows as digital experiment environments, also in connection with high performance systems (note that while the literature study to related work regarding model programming environments and methodological approaches for coupling tools has also covered approaches in other programming languages, the main findings are discussed in context of well-known implementations for the Python language). It is not only used to implement model components, but also as a glue code to connect disparate workflows with each other. The authors of [1] describe in detail how Python scripting is employed to directly link hydrodynamic and oil spill models and to reduce the manual coupling effort. A more complex approach for wrapping models in Python is presented in [2]. Thereby a whole framework for loose coupling high-fidelity multiphysics simulation models is developed, providing users with functionality to flexibly exchange and execute workflow components in parallel. In the work of [3], models are coupled by using the PyFMI package. PyFMI is adhering to the functional mockup interface, a tool-independent interface standard, that supports the exchange of models as compiled black box programs implementing a tool-independent communication interface described in an XML file. FMI model components can therefore also be imported as model components into another simulation environment to perform a co-simulation of model components. Details of the standard are further described in [4]. With PyFMI, all models that follow this standard can be integrated into one's own application, including models based on other simulation platforms like Dymola, OpenModelica, or SimulationX. Typically, in an environment using FMI, the model components are executed as a slave process which is executed and controlled by the importing simulation environment as master.

According to [5–7], the docker container technology is often used nowadays for runtime automation of complex executables and data processing workflows, since containerization can be applied as an automated runtime environment to run, control and manage different executables built with different programming languages (e.g., Python, Matlab, C, C++, Java) together with their dependencies on nodes of a computing cluster. As stated in [8,9], container virtualization technology is e.g., used for runtime automation of parallelized big data processing and analysis pipelines. Generally, a container-based architecture can be considered as a very good concept for runtime automation of complex computational workflow as well.

In [10], an architecture for ingesting data obtained from different production environments into a big data platform is presented. Apache NiFi is utilized for data ingestion to process both interval and real-time data from the production sites. NiFi allows us to setup more complex data processing workflows for data ingestion, where data arrives via a source node injecting the data into the workflow, then e.g., gets transformed and enriched in intermediate steps (NiFi process steps) and lastly is stored (e.g., in a database) via an output sink task. For creating such data processing workflow, NiFi has a well-designed, web-based workflow editor which makes it a good candidate for providing the base environment for setting up processing workflows. However, no bi-directional communication or coordination functionalities are provided for implementing large-scale co-simulation scenarios [5].

Refs. [11–15] describe different co-simulation frameworks especially developed for smart grid application scenarios. However, all these frameworks couple only two simulators and are limited to particular scenarios. They offer no possibility for the integration of many different kinds of models in order to build a more universal co-simulation platform for simulating realistic multi-domain energy system scenarios.

An implementation of a more flexible cyber-physical energy system co-simulation framework is the tool Mosaik that has been developed at the Oldenburg Institute for Information Technology [16]. It provides APIs to integrate existing simulation models and simulators into Mosaik and then combines them to build smart grid scenarios [16,17]. However, a specific implementation of the APIs by the users (e.g., writing glue code) is necessary. Moreover, as presented in [18], Mosaik can only provide the possibility to connect models and manage them in a coordinated simulation but has no support for e.g., integrating other auxiliary tools for data pre-processing or data forecasting to set up more advanced computational scientific workflows. Furthermore, as a conclusion of [19], Mosaik is suitable for entry-level, prototypical co-simulation but not for complex and extensive studies. Besides, Mosaik has no easy-to-use web user interface but a desktop user interface [20], which provides users only with a view of simulation results but no graphs of data routing, transformation and system mediation logic [5].

Ref. [21] describes an open-source "framework for network co-simulation" (FNCS) for power system transmission and distribution dynamics co-simulation. FNCS provides libraries for supporting C, Java, Matlab, Python and FORTRAN interfaces for flexible simulator integration. Besides, FNCS has no limit on the number of participating simulators and great scalability. However, users need to extend the simulators to use the interfaces for boundary buses that need to communicate with others through FNCS. No easy user interface is provided by FNCS for users to integrate their simulators and then build co-simulation scenarios easily.

Ref. [22] presents the design rationale for the hierarchical engine for large-scale Infrastructure co-simulation (HELICS), a new open-source, cyber-physical-energy co-simulation framework for electric power systems. By using HELICS, large-scale co-simulations with off-the-shelf power-system, communication, market and end-use tools can be built. However, users have to implement programming interfaces to combine their simulators with HELICS. Moreover, by using the Matlab API, HELICS can only interact with Matlab models running in Matlab environments providing a commercial Matlab license. Contrarily, the framework presented in this article can launch standalone Matlab executables in containers without the need for a Matlab license by using exported binaries and distribute them on the scalable computing cluster easily, flexibly and efficiently.

One main result of the literature study was that—at least to our knowledge—there is currently no existing solution that fulfils all the mentioned requirements for a more generic framework for setting up and managing complex computational scientific workflows. Apache NiFi, for example, implements some very good base functionalities needed for such an environment, but lacks some features for the more advanced coordination of workflow steps which is needed to set up more complex workflows. However, the literature study also shows that such an environment could be created by combining the right technologies and integrating them into a comprehensive framework.

2.2. *State-Of-The-Art Technologies*

In the following subsections, the state-of-the-art IT concepts and technologies used to implement the framework are described.

2.2.1. Microservices-Based Architecture

A microservice-based architecture is an architectural style that puts its emphasis on dividing and designing an entire application as a set of loosely coupled, lightweight, independent services [23–25]. Each independent service can be independently developed, tested, deployed, monitored, scaled and even implemented in different programming languages [23,24,26]. As the conclusion of [27–29], compared to the monolith architecture, a microservice-based architecture addresses the following concerns:

- increased resilience due to independent development and deployment of services
- fast
- better code quality
- easier debugging and maintenance
- increased productivity
- improved scalability
- freedom (in a way) to choose the implementation technology/language
- continuous delivery

Therefore, a microservice-based architecture can be a very convincing architecture style for setting up a workflow environment with the aim of loose coupling independent runnable tasks (e.g., arbitrary execution environments). Additionally, by utilizing a microservice-based architecture, the whole presented framework built as a set of separate microservices can be developed, debugged, tested, versioned, deployed and scaled easily, quickly and flexibly.

2.2.2. Container Virtualization—Docker

According to [5,29–31], compared to virtual machines, virtual containers sharing the host operating system and also reducing management overhead are lightweight and more portable. Docker is the most well-known open source application container platform. Docker utilizes resource isolation features, such as cgroups and Linux kernels to build isolated containers [5,6,8,32]. Using Docker files that describe a complete, static and executable version of an application or service including all of their other runtime dependencies, such as libraries or interpreter environments, Docker images can be created to run applications or services within Docker containers, since Docker images include all of the dependencies needed to execute applications or services within a container [6,32,33].

By using Docker, the presented framework can abstract the runtime environment of workflow tasks to Docker images that run as corresponding stand-alone executables in Docker containers on the underlying computing cluster. This contributes to a modular and loosely coupled approach. Moreover, thanks to the many benefits of Docker containers and microservices technology, the framework has high scalability and extended flexibility.

2.2.3. Container Orchestration—Kubernetes

But how can an application consisting of many microservices be managed and controlled? To do this, orchestration technologies come into play. According to [34], Kubernetes is a platform for automating deployment, scaling, and management of containerized applications. Together with Docker, Kubernetes is used to deploy applications (and therefore also microservices) in containers across clusters of computing nodes. It contains tools for orchestration, e.g., handling a set of executables as one application, service discovery (e.g., identifying instances of a certain type and checking their health for failure tolerance) and load balancing. By using Kubernetes, one or more containers can be logically associated with a pod that is the basic scheduling unit for Kubernetes to deploy an application part (e.g., consisting of one or more services having a close relationship) as a set of Docker images and manage such application parts. Pods can share resources needed by application parts across the associated containers (e.g., a certain file storage provided by a distributed file system or a certain network address). Kubernetes can automatically find a machine that has enough free computational capacity for a given pod and launches the associated containers. To avoid conflicts, each pod is assigned a unique IP address, enabling applications to use different ports under the same IP address. Kubernetes offers availability and quality checks for containers in order to heal failed containers in pods through its automatic failure recovery actions [35]. As presented in [36], compared to Docker Swarm, which is another container orchestration platform [37], Kubernetes is more powerful and provides immense scalability and automation at the same time. Additionally, the performance of Kubernetes currently surpasses that of Docker Swarm.

By using Kubernetes for the framework, containerized executables running in Pods managed by deployments on the computing cluster can be managed and controlled easily and efficiently.

2.2.4. Message Channel—Redis

Another key part of a computational scientific workflow framework is related to the question of how can the information flow between workflow tasks be organized and coordinated. Since tasks are executed in distributed containers, for coordination and message exchange between the tasks, an adequate distributed technology is needed. Such communication and coordination services are often based on key-value databases as storage (e.g., etcd [38], Zookeeper [39], et al.) and eventually use a message protocol as communication interface (e.g., Redis, REmote DIctionary Server). As presented in [40], Redis is an in-memory data structure store, used as a key-value non-relational database, cache and message broker. Due to its in-memory nature, Redis boasts high performance for read and write operations. Besides, to allow users form high-level data structures as values Redis provides five abstract data types for values: strings, lists, sets, hashes and sorted sets from which more complex objects can be constructed. Redis is written in ANSI C and lightweight with no external dependencies. Furthermore, Redis offers atomic operations for read/write actions in order to allow using data structures for coordination, e.g., for the implementation of a simple and thread-save implementation of a distributed counter for instance. Such counters will maintain consistency when manipulated by multiple parallel operating tasks.

By integrating Redis into the presented framework as a communication and coordination interface and using its practical publish/subscribe functionality, information exchange between distributed tasks and their coordination can be implemented to be very efficient, fast and failure-tolerant.

2.2.5. Data Ingestion/Transformation—Apache NiFi

As already mentioned above, NiFi as data ingestion workflow application using a data flow like task execution and programming model already provides a good set of base technologies for setting up a computational scientific workflow environment. It is an enterprise integration and dataflow automation tool that allows a user to send, receive, route, transform, and sort data [10,41]. It provides real-time control that makes it easy to manage the movement of data between any source and any destination [41]. It offers a drag-and-drop web graphical user interface for simply building and editing complex data transformation workflows by combining pre built building blocks (Apache NiFi processors) into an intuitive visual data workflow [41]. Already configured workflows can be stored, loaded and easily modified before started. The framework is highly scalable and can run on a cluster distributed on many high-performance servers [41]. Additionally, it is designed for extension that allows for building and adding own Apache NiFi processors and even allowing customizations of the key framework [5]. By using Apache NiFi as the base framework for the foreseen computational scientific workflow framework and extending it with advanced functionality for extended coordination and co-simulation interactions, the framework will allow users to easily build, execute and manage data processing or simulation workflows via an enhanced Apache NiFi web user interface [5,10]. Custom Apache NiFi processors can be designed for providing a library of custom workflow modules e.g., for integration of base models of a certain type, e.g., which are common elements for modelling smart grids [5]. Furthermore, several processors (e.g., ListenHTTP and InvokeHTTP processors) that are included in Apache NiFi library can be easily and directly used for data integration, transformation, ingestion and transmission without changing anything [5].

3. Framework Architecture

After introducing important base technologies, in this section the overall architecture and main concepts of the computational scientific workflow framework are described. The basic architecture of the framework is depicted in Figure 3. The structure of the framework consists of the following components; a web user interface based on the NiFi user interface, an extended version of the NiFi processor interface as bridge between the NiFi runtime environment and the other parts of the framework, especially the messaging communication and coordination solution based on Redis, and a NiFi external process management for encapsulating third party applications (e.g., models and other tools for performing workflow tasks) in dynamic executable containers and integration other integration logic like I/O adapters as well as e.g., file storage volumes automatically mounted by the container for easy data exchange between the encapsulated application and the workflow framework environment. In the next subchapters, the main concepts behind these parts of the framework are explained in more detail based on the role of these components.

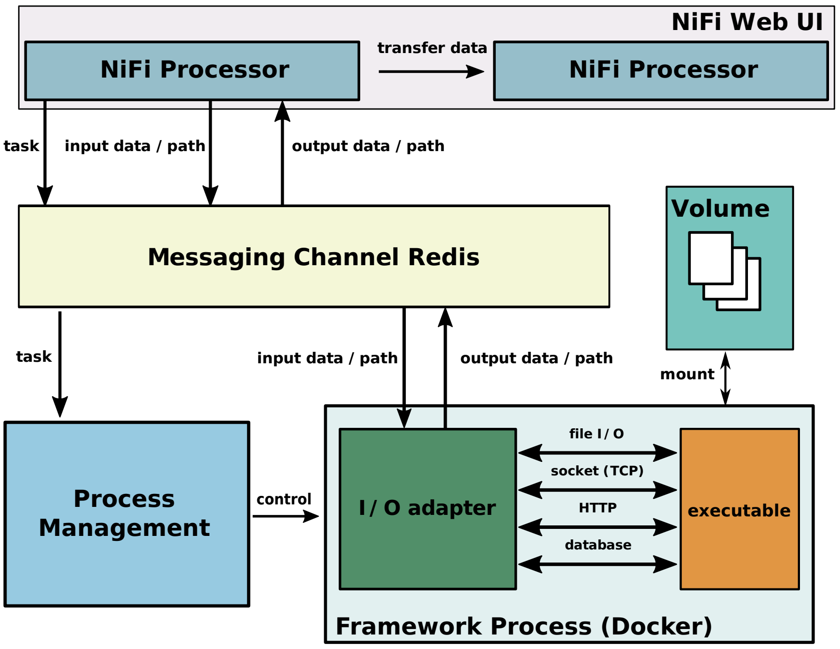


Figure 3. Architecture of the framework.

3.1. *Apache NiFi User Interface and Processors*

The framework uses the open source software Apache NiFi in order to provide a web user interface allowing users to define, operate, control and analyze complex data processing and co-simulation workflows. Apache NiFi defines a visual user interface abstraction for tasks of a Apache NiFi data flow workflow called Apache NiFi processors, that on the one hand is used by the framework to have a defined interface between the Apache NiFi software and the runtime notion of externally compiled executables which can be executed as workflow tasks. On the other hand, a NiFi processor represents a graphical element in the NiFi workflow editor palette of building blocks, which can be dragged by the editor on the pane to visually add a task to the workflow. To add further process elements to Apache NiFi, a NiFi Java processor class have to be created which extends the Nifi Processor base class and provides all information about the new processor that are needed by NiFi to integrate the new processor into the overall NiFi environment. Compiled versions of such processor classes can then be added to the NiFi processor library and will visually appear in the NiFi palette of processor elements in the workflow editor. From the NiFi processor library, standard and custom NiFi processors are reusable and can be added into a workflow using the web user interface. Furthermore, for high configuration of the framework, properties for parametrization and customization of the behaviour of a processor can be added to a NiFi processor that can be configured by the workflow editor or user via the web user interface in a configuration dialogue and are handed down to the processes and their executables at runtime. By extending the NiFi base class with own functionalities more advanced processor functionality can be easily added to the framework.

3.2. Integrating a Communication and Coordination Infrastructure Using a Message Oriented Middleware

For larger transdisciplinary, multi-domain data processing and co-simulation workflows, data exchange between executables and some coordination logic is essential. As depicted in Figure 3, a distributed messaging infrastructure is integrated into the framework for allowing communication between the framework processes and providing means for coordination. Redis and Apache Kafka, both of them having efficient and very scalable messaging functionality, can be utilized as a communication interface for the framework and were tested for providing the message functionality.

As Table 1 shows, Redis is a bit different from Apache Kafka in terms of its storage and various functionalities. As mentioned above, the framework requires fast real-time data processing with low communication or I/O overhead. At its core, Redis is an in-memory data store that uses its primary memory for storage and processing which makes it much faster than the disk-based Apache Kafka. Beside this, Redis can be used as a high-performance database, and a message broker, which better matches the framework's requirements, since most of the data in the framework needs to be transferred quickly, and it is not necessary to store the data persistently on a hard disk. Redis features publish/subscribe messaging which is push-based while the publish/subscribe messaging of Apache Kafka is pull-based. That means that messages published to Redis will be automatically delivered to subscribers instantly, while in Apache Kafka data/messages are never pushed out to consumers; the consumer will ask for messages when the consumer is ready to handle the message. In the case that consumers do not process data fast enough, Apache Kafka will have to read from a disk and not from memory which will slow down its performance. Moreover, In most cases, the framework deals with small short lived messages with a minimal latency. Redis, using an in-memory storage can store the small messages within the limit of memory allocated to Redis. However, since the memory storage used by Redis is typically smaller than a disk, it is necessary to clear it regularly and making room for new data.

Table 1. Comparison of framework requirements to the functionalities of Apache Kafka and Redis.

Framework Requirements	**Apache Kafka**	**Redis**
fast storage	in-memory and disk-based	in-memory
a database and a message broker	a message service	can be used as a database, cache, and message broker
short lived messages	can keep flushing data for longer period of time	retain data for short period of time
small amount of data	large data size	small data size

Therefore, in summary, compared to Apache Kafka, as a generic message-oriented in-memory-based communication interface, Redis is a better solution for the framework to establish communication between the executables and the *I/O* adapters (see Figure 3).

3.3. Process Management

The process management is used to orchestrate the containers that are started and stopped dynamically by the framework. With the help of Kubernetes as a container-orchestration system on a cluster, the process management service is running as a master deployment for automating, scaling and maintaining other containerized processes that are themselves running as Kubernetes deployments.

3.4. Framework Processes

To support the execution of workflows in the framework, all executables and its dependencies (e.g., solvers, libraries and runtime environments) have to be integrated into different Docker images for later use. Moreover, a model description including all essential commands required to run the executable should be given by the users and automatically converted into a Python script. Additionally, for each executable, appropriate I/O adapters are used for instrumenting the data exchange between the executables. As shown in Figure 3, various I/O adapters, such as file adapter, TCP adapter, HTTP adapter, are offered by the framework. Finally, by using a Dockerfile, an executable, an I/O adapter, a Python script and the relevant dependencies are integrated into one Docker image. Based on the image and Kubernetes, the process management mentioned in the previous section can run Docker containers to perform the executables.

While this procedure suggests that there is a large overhead in integrating new models and tools into the environment, the benefits of this procedure for the scientists far outweigh the additional effort. Once the integration is done, the scientist itself does not have to care about any of the runtime aspects for performing the workflow task (e.g., setting up a computer so that the application can be run successfully, e.g., sometimes there are libraries missing or a patch of the operating system breaks the application, starting and stopping the application, controlling resource usage). As mentioned in Sections 2.2.1 and 2.2.2, by leveraging Docker container and microservices, programs can be easily and flexibly executed in Docker containers running within different independent simulation nodes on a computing cluster, and the end users do not have to care about on which computing nodes the tasks of the workflow get started. Every aspect of the workflow can be supervised by him using a web browser for accessing the NiFi user interface. With Redis and the I/O adapters provided by the framework, data exchange between executables can be easily instrumented without requiring users to change their executables or implement interfaces and adapters.

3.5. Volume

Of course, for some special use cases, large chunks of data also need to be exchanged between tasks that could not easily be transferred via the Redis message queues. As presented in [5] and Figure 3, for such cases Docker containers could mount—as already mentioned—file system volumes that are provided by a distributed file system and shared by all containers which utilize these to solve the storage and transfer large batches of data as files. Since Kubernetes supports multiple distributed file systems for providing mounted volumes for different purposes (https://kubernetes.io/docs/concepts/storage/volumes/), different volume types are available for the framework to improve the speed of reading and writing large files. The efficiency of reading and writing large files directly from such a volume is much higher than the transmission of large files with Redis. Especially, when Redis has not allocated enough memory. The I/O adapters automatically detect if there are large files to transfer. In that case, the adapters will only transmit the unique IDs and not the original entire file. According to the IDs, the next processor in the workflow and its Docker container can use its I/O adapter to read the data of the large file from the volume directly and efficiently.

3.6. Parallelization and Synchronization

For maximizing the performance of certain tasks within a workflow, the framework implements a concept for running multiple instances of a workflow task (e.g., a NiFi processor) in parallel. In many use cases the results of parallel computations have to be aggregated and synchronized for further use. Therefore, as depicted in Figure 4, the framework provides a coordination service in order to not only allow parallel data processing and execution of a processor but also to fork and join data streams. To describe this concept more mathematically, a dataflow (sort of data stream) can be defined as ordered sequence of data elements:

$$D_{flow}(i) = V_i, i \in Z^+, V \in M, \tag{1}$$

where:

Z^+ = set of ordered indexes, always positive,
M = set of possible (structured) data values.

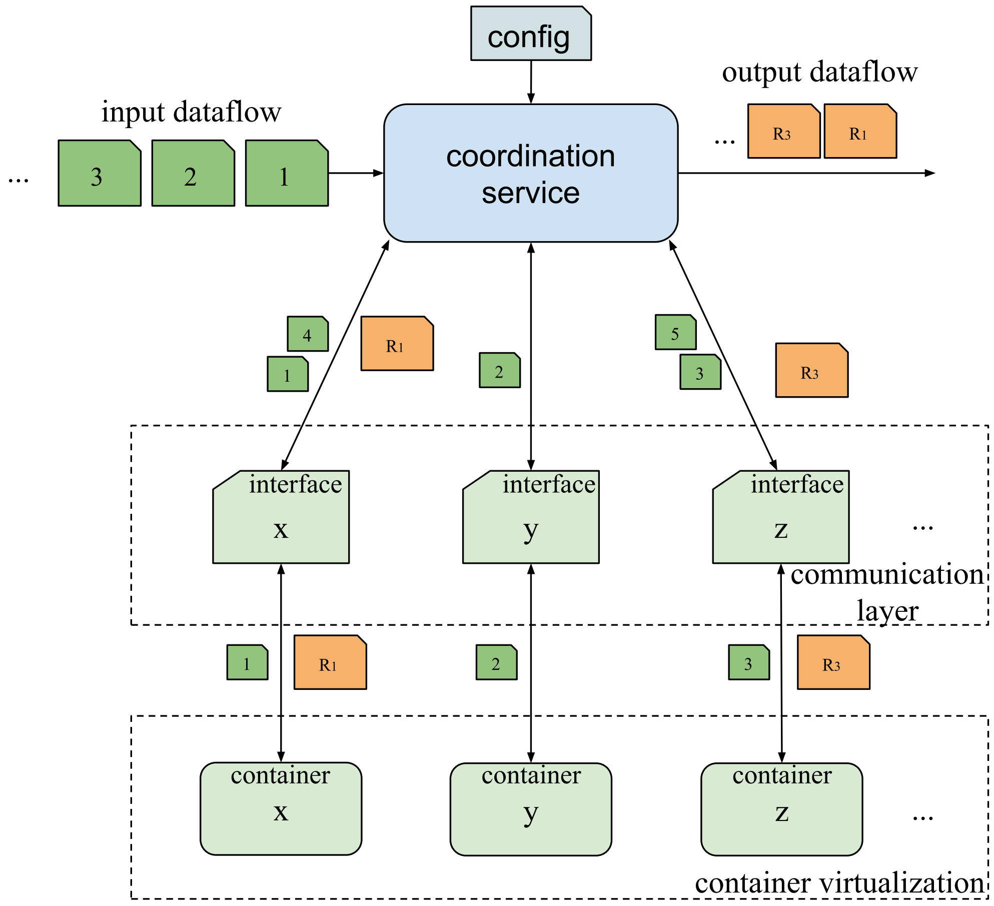


Figure 4. Example of the parallelization and coordination concept with unsorted result data.

A common form of parallel dataflow processing would now be to distribute the elements of a dataflow to a set of parallel instances of a workflow task such that each instance of the task carries out the same computation on different elements of the dataflow concurrently (e.g., fork the dataflow). Even in this scenario, many solutions for distributing the data elements to the different processor instances are possible, so let us take a simple "worker pool"-like distribution where each data element is scheduled to be processed by the next free task instance. For implementing such a dataflow processing strategy in the presented framework, first the number of concurrent tasks and the type of distribution (e.g., "worker pool") has to be defined in the config interface of a NiFi processor by the workflow editor. Then, an executable coordinator, which is located in each NiFi processor as part of the abstract processor class, will launch the given number of parallel working instances in a corresponding number of separate containers. Additionally, the "work pool"-based distribution of the dataflow elements will be instrumented in the communication layer. Therefore, as shown in Figure 4, a dataflow D_{flow} is transferred to the executable coordinator and stored in a queue for distribution. By using the different communication channels and the distribution strategy, the executable coordinator distributes data elements to the task instances, e.g., $D_{flow}(1)$, $D_{flow}(2)$ and $D_{flow}(3)$ are handled in parallel by the task instances in the containers x, y and z. Once one instance finishes its data processing, the corresponding result is transmitted back to the coordinator, and the instance will obtain the next input data from the queue, e.g., as presented in Figure 4, after the result data R_1 and R_3 arrives at the coordinator, the next input data $D_{flow}(4)$ and $D_{flow}(5)$ are transferred to the two available instances in containers x and y. If there are no further processing requirements for the sequence of output data, the coordinator will pass the received result data to the next NiFi processor (this strategy is depicted in Figure 4). The results will be collected into an output data flow (join operation) which is normally sorted based on availability of the results. If the result data have also to be sorted in the same order as the input dataflow defined by the workflow editor ("keep order"), the coordinator will sort the result data sequentially based on the index i of the input elements. In addition, if for one dataflow, the NiFi processor getting the output dataflow requires that all parallel processed result data must be available as one large result data set, the coordinator will wait until all result data arrive completely, then sort and join them together as one large result data set according to the indices i, and finally transfer it.

Another dataflow processing option implemented by the framework would be that the coordinator can also join multiple incoming dataflows into one output result dataflow by somehow combining dataflow elements with the same index. As demonstrated in Figure 5, the coordinator would obtain, in this case, two dataflows $L_{flow}(i)$ and $P_{flow}(i)$ in parallel. Each element of $L_{flow}(i)$ has to be combined with its corresponding element in $P_{flow}(i)$. Whenever elements arrive in the queues belonging to the dataflows, the coordinator will check if all corresponding elements already exist. Then they will be transferred together to a task instance that calculates a result which will be placed in the output queue. If not all pairing input elements are available, the arriving elements will be stored until a complete tuple is available. In conclusion, the coordinator can perform quite different strategies to distribute input data to the task instances and to collect and manage the corresponding results without causing data corruption or loss.

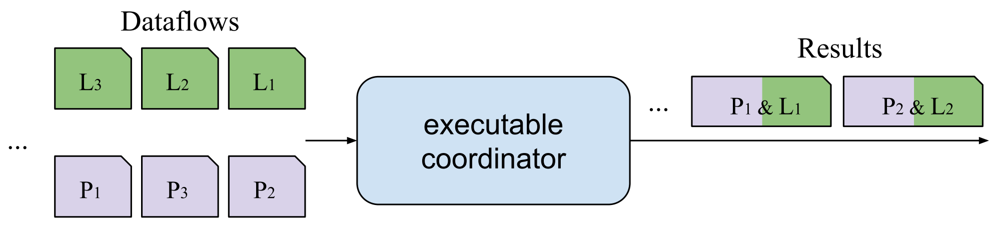


Figure 5. Example for the synchronization of multiple incoming dataflows.

The above strategies for coordination and dataflow processing were shown as example strategies, since they are used by the concrete example workflow for benchmarking the parallel execution performance of the framework as discussed further below. Many more dataflow coordination operations are possible and new ones could be easily implemented in future versions of the framework.

4. Results

To evaluate the behaviour and performance of the presented framework regarding parallel processing as discussed in the previous chapter, a common use case within Energy Systems Analysis benefitting from parallelization was used and executed with different numbers of parallel task instances. The use case workflow performed calculations for optimally dimensioning a cross-regional energy system, taking conventional and renewable energy sources as well as various storage, conversion, and transportation technologies into account. It can be parallelized by e.g., splitting a large geographical area into separate areas where calculations can be performed in parallel. Based on spatially and temporally resolved weather data and detailed market and techno-economic information, the total installed capacities, costs of energy carriers, amounts of power generated, total energy demand, and total system costs were determined. The workflow used for testing the framework included four Python-based energy systems models. The energy demand model estimated the total energy demand for a given region and year, based on population projections and statistics for energy consumption within the mobility sector. The land eligibility model processed socio-technical parameters to determine suitable areas for renewable energy production. The turbine placement and turbine simulation models selected specific sites for wind turbines, and simulated respective time-dependent production rates. Furthermore, the workflow included Conversion 1 and Conversion 2, two executables that arranged the data as an appropriate input format for a subsequent optimization model. As displayed in Figure 6, the workflow was divided into two independent sub-workflows, which had to be executed for each region considered. ListenHTTP was utilized as a HTTP server for inputs. The coordination service synchronized the outputs from the Conversion 1 and Conversion 2, which is described in Section 4.2. For optimal performance, the whole procedure should be calculated in parallel. Further information on the models and their integration into a more sophisticated model chain can be found in [42–47].

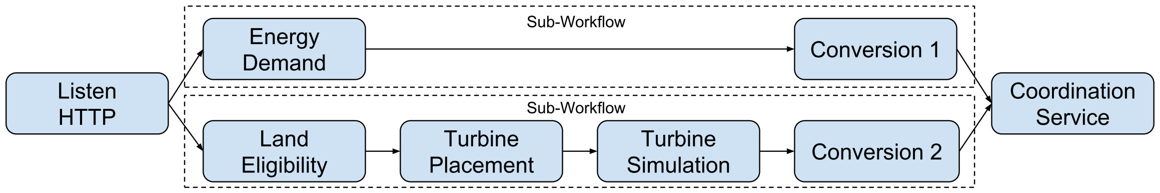


Figure 6. Use case: model workflow for designing a cross-regional energy system.

4.1. *Benchmarking*

To discover the performance of the framework while executing the example workflow, several benchmarks for measuring memory usage, execution time comparing to manual execution, and overhead for parallel tasks were performed during the execution of the workflow. As mentioned in Section 3.5, since multiple file systems for different purposes were provided by Kubernetes, the I/O performance of the framework depended also on the underlying file system of the mounted volumes (however, within the evaluation discussed in the present article, an additional benchmark to assess the I/O performance of the chosen filesystem type was not performed in view of the compactness of the paper).

4.1.1. Memory Usage

The memory usage with and without the framework required for the same workflow are shown in Figures 7 and 8, respectively. Compared to the manual sequential execution of each model without the framework, six containers, including one executable, each ran at the same time in the framework, were used in the framework setup, which required more total memory than a manual sequential execution, obviously. The memory usage was monitored for the manual execution and for each container within the framework. However, both resulting curve trends look similar, especially after the first 45 min for executing the models, presented above. In land eligibility, turbine placement, energy demand and Conversion 1, nearly 2.8 GiB total memory was used to run the turbine simulation for both situations. This means that the execution in the container environment did not demand any significant additional memory when performing the task. Beside this, in both cases the execution time of a task instance in a single container was also almost the same as in the standalone case. Thus, if the CPU power provided by the container was comparable to the CPU power provided for the standalone execution of the same executable, then the processing time was equal. Thus, the usage container technology did not add a significant overhead to CPU usage. Therefore, the framework required almost the same memory footprint for executing an executable as needed for a manual execution, and the performance for running single model instances was nearly equivalent in both cases. Clearly, the execution time also depended on the performance of the computing node itself which could be assumed as similar for this comparison.

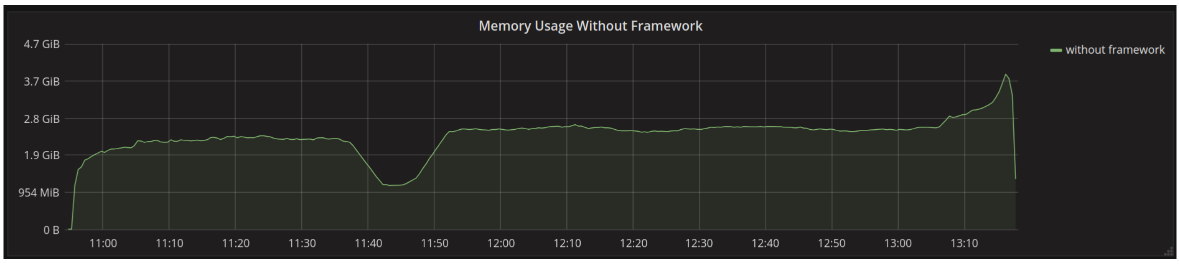


Figure 7. Memory usage without the framework run as single executable.

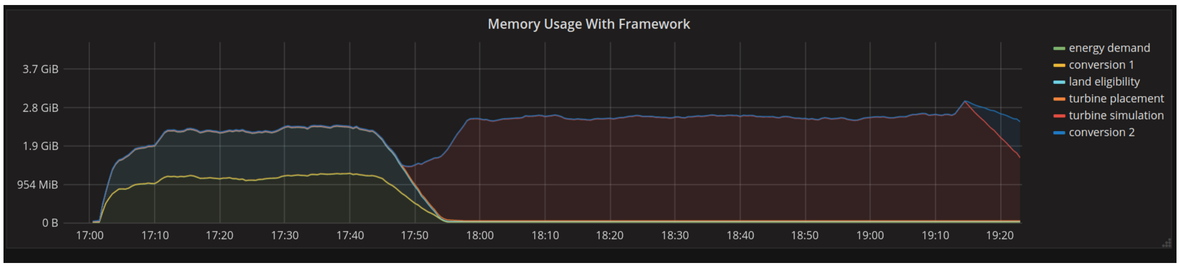


Figure 8. Memory usage of the different processes within the framework.

4.1.2. Execution Time

Table 2 lists the manual execution time and the execution time with the framework required for each model as well as the total execution time for all models. It is obvious from the values, that each model needed basically the same execution time in both runtime environments. Manual execution without the framework for all models took about *238.903* s, while the model's execution with the framework took about *239.591* s. Thanks to the container-based virtualization technology on the underlying cluster computing environment and the microservice-based architecture of the framework, the difference in both model execution time was almost negligible. Therefore, it can be concluded that the model running environment and performance provided by the framework for single instances are basically the same as the manual ones.

Table 2. Comparison of execution time with and without the framework.

Model	Manual Execution Time (s)	Execution Time with Framework (s)
land eligibility	30.895	31.614
turbine placement	5.245	5.147
turbine simulation	180.357	178.956
conversion 2	22.406	23.874
energy demand	30.988	30.615
conversion 1	0.504	0.517
all models	238.903	239.591

In order to analyze the communication and workflow processing overhead added by the framework, 20 tests with different parameters wew performed.

In Figure 9, the horizontal axis and the vertical axis respectively indicate the whole execution time, t, of each test and the ratio of the overhead, o, to the whole execution time generated by each test, namely o/t. As can be derived from the curve in Figure 9, the proportion of overhead in the whole process execution time was gradually reduced. Compared to the entire execution time, this little overhead can be easily outweighed by the performance advantages of parallel computing, which will be discussed in the next section.

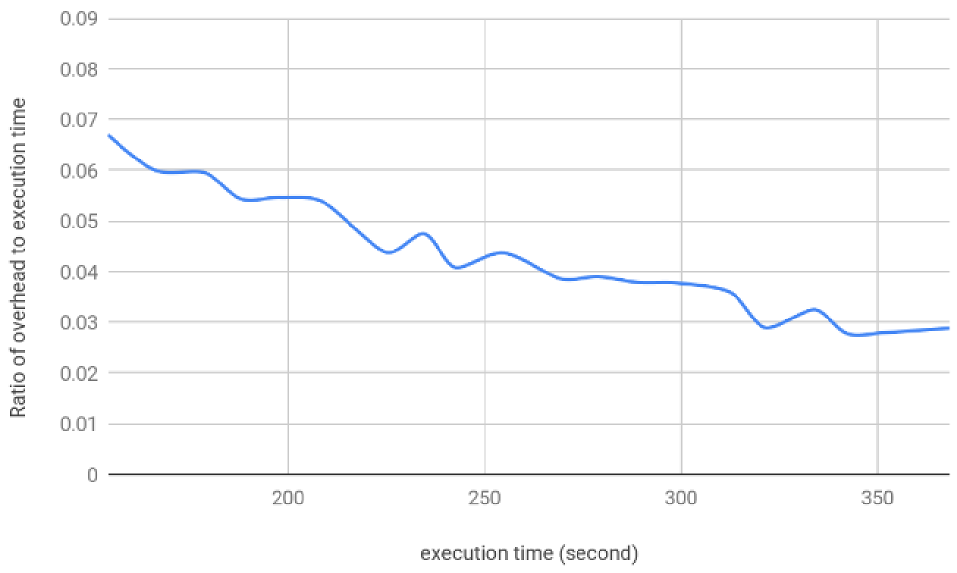


Figure 9. Ratio of overhead to whole process time.

4.1.3. Parallelization

In order to study the scalability of the framework, 10 different parallel tests were completed. In Figure 10, the number of parallel tasks and execution time for each test represent the horizontal and vertical axes, respectively. For each test, the amount of executed tasks was equal to the amount of inputs sent to the workflow. The expected result was a constant curve since each task is processing one input. As shown in Figure 10, as the number of parallel tasks increased from one to ten, the overall execution time of each parallel tasks increased slightly. Relative to the overall execution time (156 s), the small increase (about 8 s) due to the parallelism is acceptable and almost negligible. Additionally, the parallelism of only two tasks can already overcome the small overhead mentioned in the previous section, since the execution of two tasks in sequence took about 310 s, while the time of executing two tasks in parallel with the framework still remained about 156 s. The parallelism provided by the framework can be applied to many use cases to significantly save runtime and improve operational efficiency, especially for complex tasks such as iterative grid optimization. Therefore, from the tests' results it can be concluded that the framework has high scalability and extended flexibility.

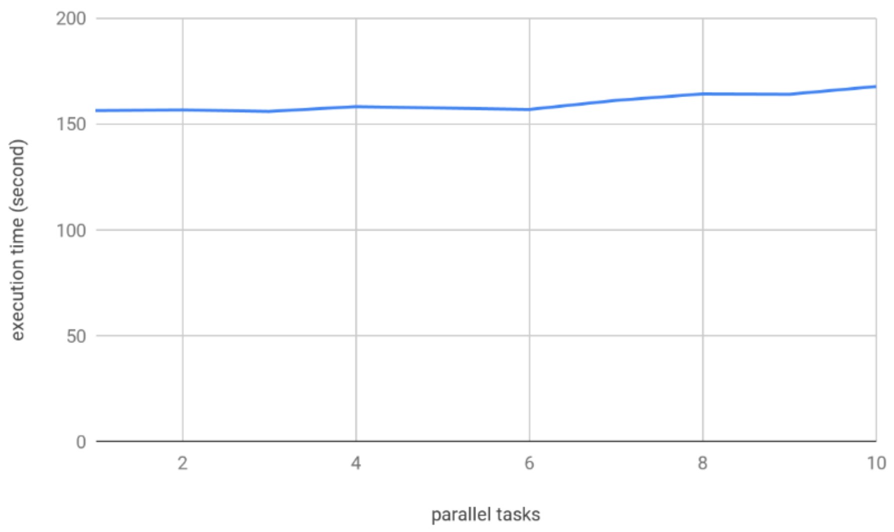


Figure 10. Execution time for multiple parallel tasks.

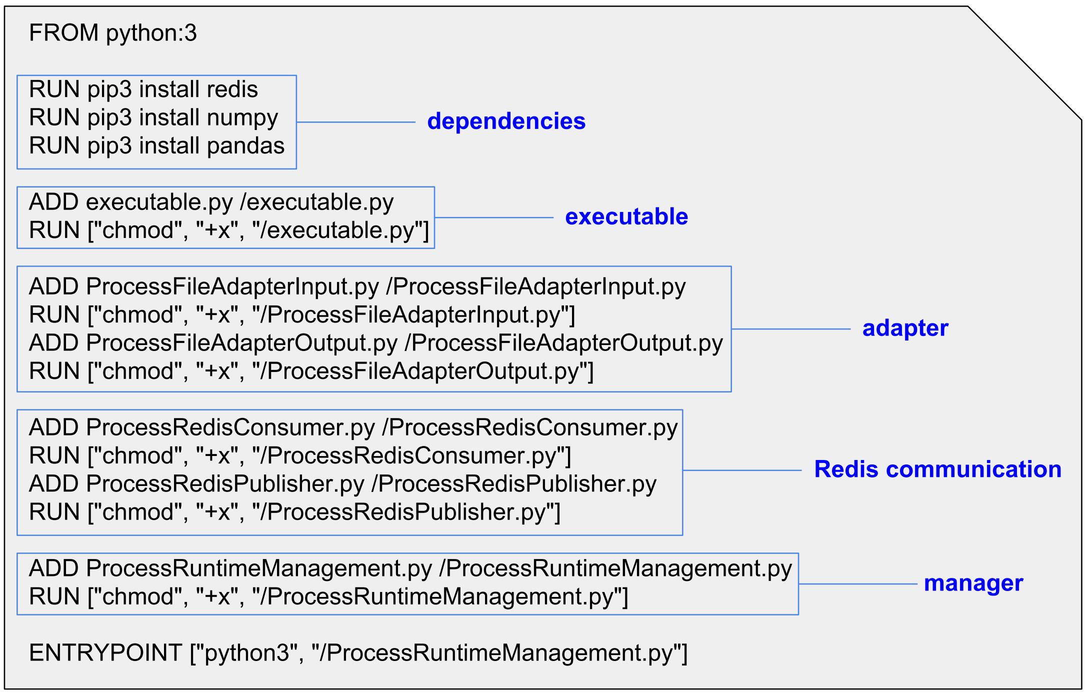
4.2. Evaluation of the Coordination Service for Synchronization of Multiple Inputs

To synchronize multiple inputs from the Apache NiFi processors Conversion 1 and Conversion 2 the coordination service shown in Figure 6 utilized the implementation of the coordination functionality described in Section 3.6 for the energy systems models workflow. Each input had its own regionID. The first input from the Conversion 1 was always transferred faster than the input from the Conversion 2, which means that the coordination service could receive several inputs from Conversion 1 but none from Conversion 2. Thus, the execution of the next task had to wait until a pair of inputs was available. When an input from the Conversion 2 was obtained by the coordination service, according to the regionID, the input data from Conversion 1 can be extracted. Thereafter, the pair of input data was passed for the next step. This synchronization overhead means that the example workflow cannot scale linearly with the number of parallel instances, but beside this necessary synchronization overhead, the scalability of the workflow processing by using more parallel instances of tasks was quite good.

5. Discussion

In addition to the discussion of the observed runtime behavior of the presented framework, in this section also the architectural design decisions are considered as they have significant impact on the software domain processes within the workflow development cycle of computational science like Energy Systems Analysis (compare Figure 1). A major aspect in this regard is, that the usage of the container virtualization employed by the framework allows the integration of all kinds of models and other executables, regardless of the operating system or the used programming languages. As long as all dependencies needed for execution are provided, Docker files and images of any application or service can be built and embedded into a workflow. By using the framework, it therefore is no longer necessary that already existing models—while meeting the corresponding scientific requirements—will have to be redeveloped, only because they cannot be integrated into a research organization's software infrastructure. As a result, a considerable amount of development and implementation effort can be saved with each complex model that can additionally be treated as a black box and does not have to be reverse engineered. However, if no appropriate model is available, a new model still has to be developed from scratch. But also in this case the chosen architecture is an asset, indirectly allowing a high degree of freedom for choosing the technology stack for model implementation, since all programming languages, technology environments, and resources can be used within the software implementation process.

Besides the possibility to integrate the most diverse executables, the involved effort has to be discussed in order to describe the usefulness of the framework for supporting computational research processes. As explained in Sections 3.1 and 3.4, a Docker file has to be written to make an executable applicable in line with the framework. As exemplary shown in Figure 11, this includes in detail the listing of all program libraries and other software components the executable is depending on. Then the source code file of the executable is named and an individual set of adapters, which themselves come with the framework, is selected. The communication and management setup is the same for all executables and does not have to be modified by the user. Afterwards, with a single instruction in the command line, a corresponding Docker image is built, which then can easily be integrated into a workflow via the Apache NiFi user interface. This short setup process can be completed in minutes even by users with little programming experience, regardless of whether the executable is an in-house development. Requiring interface knowledge only, the framework performs significantly better in model integration than using a more tight coupling approach, which always involves sophisticated and time-consuming code base alterations. Also, a loose coupling approach—which is on the one hand typically based on an individually programmed workflow coupling script or on a kind of workflow or co-simulation-based framework and on the other hand far more inflexible in terms of integrating different technologies—requires at least the same amount of integration effort as using the presented framework. Furthermore, the framework offers the advantage that the integration effort only needs to be done once per model, not once per workflow like in the tight and loose coupling approaches of other frameworks, since the created Docker images and the NiFi processors are stored in a Docker registry and in a NiFi processor library respectively and can be reused in all future iterations of the workflow development cycle, which is particularly beneficial in the dynamic field of computational research.



```
FROM python:3

RUN pip3 install redis
RUN pip3 install numpy
RUN pip3 install pandas

ADD executable.py /executable.py
RUN ["chmod", "+x", "/executable.py"]

ADD ProcessFileAdapterInput.py /ProcessFileAdapterInput.py
RUN ["chmod", "+x", "/ProcessFileAdapterInput.py"]
ADD ProcessFileAdapterOutput.py /ProcessFileAdapterOutput.py
RUN ["chmod", "+x", "/ProcessFileAdapterOutput.py"]

ADD ProcessRedisConsumer.py /ProcessRedisConsumer.py
RUN ["chmod", "+x", "/ProcessRedisConsumer.py"]
ADD ProcessRedisPublisher.py /ProcessRedisPublisher.py
RUN ["chmod", "+x", "/ProcessRedisPublisher.py"]

ADD ProcessRuntimeManagement.py /ProcessRuntimeManagement.py
RUN ["chmod", "+x", "/ProcessRuntimeManagement.py"]

ENTRYPOINT ["python3", "/ProcessRuntimeManagement.py"]
```

Figure 11. Structure of a Dockerfile for the creation of a custom workflow using the presented framework.

Another major characteristic is that, unlike the other coupling approaches, the presented framework does not require any additional programming effort to automatically execute multiple instances of a workflow or parts of a workflow in parallel, since this coordination functionality is already inherent to the NiFi processor logic described in Section 3.6. The benchmark results presented in Section 4.1 show that the induced memory overhead is almost negligible and the slight increase in runtime is easily compensated by utilizing the parallel computation capabilities. At the same time, as described in Sections 3.6 and 4.2, the architecture includes generic functionality to synchronize results from parallel processed calculations, which allows the supplementation of workflows with components to aggregate, consolidate, and prepare result data for evaluation and interpretation, tasks necessary at the end of every scientific experiment. The combination of those both features prevents, that the logic to run subprocesses and entire workflows in parallel as well as the communication control and consistency assurance mechanisms have to be designed, implemented, and tested with each newly drafted workflow.

It is the authors opinion that all the discussed potentials for saving time and effort in the technical coupling of executables to workflows and the decrease of required software development skills easily compensate the amount of effort that has to be made for installing and learning the framework, and that the presented approach can significantly help streamlining the workflow development processes of an institution performing research within the domain of computational science. Once the framework is set up in an institution and the models and auxiliary tasks are instrumented as NiFi processors within the workflow platform, all scientists of an organization can directly benefit from the reuse of components, which are immediately available within the platform.

6. Conclusions

The efficient automation and parallel computation of complex workflows are of increasing importance for performing computational science. When coupling heterogeneous models and other executables, a wide variety of software infrastructure requirements must be considered to ensure the compatibility of workflow components. The consistent utilization of advanced computing capabilities and the implementation of sustainable software development concepts that guarantee maximum efficiency and reusability are further issues that must regularly be met by scientists within research organizations. This article addresses these challenges by presenting a generic, modular and highly scalable process operation framework for efficient coupling and automated execution of complex computational scientific workflows. By implementing a microservice architecture that utilizes Docker container virtualization and Kubernetes container orchestration, the framework supports the flexible and efficient parallelization of computational tasks on distributed cluster nodes. Additionally, the use of Redis and different I/O adapters offer a scalable and high-performance communication infrastructure for data exchange between executables, allowing the computation of workflows without requiring the adjustment of executables or the implementation of interfaces or adapters. The specification, processing, controlling, and evaluation of computational scientific workflows is ensured by a convenient and easily understandable user interface, which is based on Apache NiFi technology. By implementing and executing a complex Energy Systems Analysis workflow, the performance of executing workflows with the presented framework was evaluated. The memory footprint for running an executable using the framework is similar to a manual execution. Moreover, the performance and running environment for single model instances of the framework are also nearly equivalent. Due to the high scalability and extended flexibility of the framework, use cases benefitting from parallel execution can be parallelized thereby significantly saving runtime and improving operational efficiency, especially during complex tasks like iterative grid optimization.

In order to consolidate the results presented in this article and to further verify the usefulness of the framework, the next step is to broaden the user base and gain experience with additional use cases. As part of the further development of the presented framework, the aim is to provide users with the opportunity to graphically depict workflow information and to enhance data management and examination capabilities. Furthermore, a more comprehensive user interface will be implemented to allow the upload and automated integration of customized executables into the framework.

Author Contributions: Individual contributions of each of the authors to the research are the following: conceptualization, J.L. and E.B.; methodology, J.L.; resources, P.K. and D.S.R.; software and formal analysis, J.L.; supervision, C.D., M.R., D.S. and V.H.; validation, J.L., E.B., P.K. and D.S.R.; writing—original draft, J.L., E.B., P.K. and D.S.R.; writing—review & editing, J.L., E.B., P.K., D.S.R., C.D., M.R. and V.H.

Acknowledgments: This work was supported by the Helmholtz Association under the Joint Initiative "Energy System 2050—A Contribution of the Research Field Energy".

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Rosenzweig, I.; Hodges, B.R. *A Python Wrapper for Coupling Hydrodynamic and Oil Spill Models*; Center for Research in Water Resources, University of Texas at Austin: Austin, TX, USA, 2011.
2. Foley, S.S.; Elwasif, W.R.; Bernholdt, D.E. The Integrated Plasma Simulator: A Flexible Python Framework for Coupled Multiphysics Simulation. In *PyHPC 2011: Python for High Performance and Scientific Computing*; Available as Oak Ridge National Laboratory Technical Report ORNL/TM-2012/57; Oak Ridge National Laboratory: Oak Ridge, TN, USA, 2011.
3. Andersson, C.; Åkesson, J.; Führer, C. *PyFMI: A Python Package for Simulation of Coupled Dynamic Models with the Functional Mock-up Interface*; Technical Report in Mathematical Sciences; Centre for Mathematical Sciences, Lund University: Lund, Sweden, 2016; Volume 2016, No. 2.
4. Blochwitz, T.; Otter, M.; Arnold, M.; Bausch, C.; Clauß, C.; Elmqvist, H.; Junghanns, A.; Mauss, J.; Monteiro, M.; Neidhold, T.; et al. The functional mockup interface for tool independent exchange of simulation models. In Proceedings of the 8th International Modelica Conference, Technical University, Dresden, Germany, 20–22 March 2011; No. 063; Linköping University Electronic Press: Linköping, Sweden, 2011.

5. Liu, J.; Braun, E.; Düpmeier, C.; Kuckertz, P.; Ryberg, D.S.; Robinius, M.; Stolten, D.; Hagenmeyer, V. A Generic and Highly Scalable Framework for the Automation and Execution of Scientific Data Processing and Simulation Workflows. In Proceedings of the 2018 IEEE International Conference on Software Architecture (ICSA), Seattle, WA, USA, 30 April–4 May 2018; pp. 145–155.
6. Naik, N. Docker container-based big data processing system in multiple clouds for everyone. In Proceedings of the 2017 IEEE International Systems Engineering Symposium (ISSE), Vienna, Austria, 11–13 October 2017; pp. 1–7.
7. Liu, J.; Dupmeier, C.; Hagenmeyer, V. A new concept of a generic co-simulation platform for energy systems modeling. In Proceedings of the FTC 2017—Future Technologies Conference 2017, Vancouver, BC, Canada, 29–30 November 2017; pp. 97–103.
8. Merkel, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.* **2014**, *2014*, 239.
9. Anderson, C. Docker [Software engineering]. *IEEE Softw.* **2015**, *32*, 102-c3. [CrossRef]
10. Sarnovsky, M.; Bednar, P.; Smatana, M. Data integration in scalable data analytics platform for process industries. In Proceedings of the 2017 IEEE 21st International Conference on Intelligent Engineering Systems (INES), Larnaca, Cyprus, 20–23 October 2017; pp. 000187–000192.
11. Amarasekara, B.; Ranaweera, C.; Nirmalathas, A.; Evans, R. Co-simulation platform for smart grid applications. In Proceedings of the 2015 IEEE Innovative Smart Grid Technologies—Asia (ISGT ASIA), Bangkok, Thailand, 3–6 November 2015; pp. 1–6.
12. Sun, X.; Chen, Y.; Liu, J.; Huang, S. A co-simulation platform for smart grid considering interaction between information and power systems. In Proceedings of the ISGT 2014, Washington, DC, USA, 19–22 February 2014; pp. 1–6.
13. Bian, D.; Kuzlu, M.; Pipattanasomporn, M.; Rahmanm, S.; Wu, Y. Real-time co-simulation platform using OPAL-RT and OPNET for analyzing smart grid performance. In Proceedings of the 2015 IEEE Power & Energy Society General Meeting, Denver, CO, USA, 26–30 July 2015; pp. 1–5.
14. Bhor, D.; Angappan, K.; Sivalingam, K.M. A co-simulation framework for Smart Grid wide-area monitoring networks. In Proceedings of the 2014 Sixth International Conference on Communication Systems and Networks (COMSNETS), Bangalore, India, 7–10 January 2014; pp. 1–8.
15. Shu, D.; Xie, X.; Jiang, Q.; Guo, G.; Wang, K. A Multirate EMT Co-Simulation of Large AC and MMC-Based MTDC Systems. *IEEE Trans. Power Syst.* **2018**, 33, 1252–1263. [CrossRef]
16. Schuette, S.; Scherflke, S.; Troeschel, M. Mosaik: A framework for modular simulation of active components in smart grids. In Proceedings of the 2011 IEEE First International Workshop on Smart Grid Modeling and Simulation (SGMS), Brussels, Belgium, 17 October 2011; The Series Lecture Notes in Computer Science, pp. 55–60.
17. Palensky, P.; Van Der Meer, A.A.; Lopez, C.D.; Joseph, A.; Pan, K. Cosimulation of Intelligent Power Systems: Fundamentals, Software Architecture, Numerics, and Coupling. *IEEE Ind. Electron. Mag.* **2017**, *11*, 34–50. [CrossRef]
18. Schloegl, F.; Rohjans, S.; Lehnhoff, S.; Velasquez, J.; Stein-brink, C.; Palensky, P. Towards a classification scheme for cosimulation approaches in energy systems. In Proceedings of the 2015 International Symposium on Smart Electric Distribution Systems and Technologies (EDST), Vienna, Austria, 8–11 September 2015; pp. 516–521.
19. Steinbrink, C.; van der Meer, A.A.; Cvetkovic, M.; Babazadeh, D.; Rohjans, S.; Palensky, P.; Lehnhoff, S. Smart Grid Co-Simulation with MOSAIK and HLA: A Comparison Study. *Comput. Sci. Res. Dev.* **2018**, *33*, 135. [CrossRef]
20. Easy-to-Use GUI for Mosaik Available. Available online: https://mosaik.offis.de/2017/06/21/maverig_installation/ (accessed on 3 January 2019).
21. Huang, R.; Fan, R.; Daily, J.; Fisher, A.; Fuller, J. Open-source framework for power system transmission and distribution dynamics co-simulation. *IET Gener. Transm. Distrib.* **2017**, *11*, 3152–3162. [CrossRef]
22. Palmintier, B.; Krishnamurthy, D.; Top, P.; Smith, S.; Daily, J.; Fuller, J. Design of the HELICS high-performance transmission-distribution-communication-market co-simulation framework. In Proceedings of the 2017 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES), Pittsburgh, PA, USA, 21 April 2017; pp. 1–6.

23. Hasselbring, W.; Steinacker, G. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, 5–7 April 2017; pp. 243–246.
24. Dragoni, N.; Giallorenzo, S.; Lafuente, A.L.; Mazzara, M.; Montesi, F.; Mustafin, R.; Safina, L. Microservices: Yesterday Today and Tomorrow. In *Present and Ulterior Software Engineering*; Springer: Cham, Switzerland, 2017; pp. 195–216.
25. Microservices. Available online: https://microservices.io/ (accessed on 3 January 2019).
26. Singh, V.; Peddoju, S.K. Container-based microservice architecture for cloud applications. In Proceedings of the 2017 International Conference on Computing, Communication and Automation (ICCCA), Greater Noida, India, 5–6 May 2017; pp. 847–852.
27. Khazaei, H.; Barna, C.; Beigi-Mohammadi, N.; Litoiu, M. Efficiency Analysis of Provisioning Microservices. In Proceedings of the 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Luxembourg, 12–15 December 2016; pp. 261–268.
28. Villamizar, M.; Garcés, O.; Ochoa, L.; Castro, H.; Salamanca, L.; Verano, M.; Casallas, R.; Gil, S.; Valencia, C.; Zambrano, A.; et al. Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. In Proceedings of the 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Cartagena, Colombia, 16–19 May 2016; pp. 179–182.
29. Amaral, M.; Polo, J.; Carrera, D.; Mohomed, I.; Unuvar, M.; Steinder, M. Performance evaluation of microservices architectures using containers. In Proceedings of the 2015 IEEE 14th International Symposium on Network Computing and Applications (NCA), Cambridge, MA, USA, 28–30 September 2015; pp. 27–34.
30. Soltesz, S.; Pötzl, H.; Fiuczynski, M.E.; Bavier, A.; Peterson, L. Container-based operating system virtualization: A scalable highperformance alternative to hypervisors. *ACM SIGOPS Oper. Syst. Rev.* **2007**, *41*, 275–287. [CrossRef]
31. Felter, W.; Ferreira, A.; Rajamony, R.; Rubio, J. An updated performance comparison of virtual machines and linux containers. *Technology* **2014**, *28*, 32.
32. Docker. Available online: https://www.docker.com/ (accessed on 3 January 2019).
33. Li, Y.; Xia, Y. Auto-scaling web applications in hybrid cloud based on docker. In Proceedings of the 2016 5th International Conference on Computer Science and Network Technology (ICCSNT), Changchun, China, 10–11 December 2016; pp. 75–79.
34. Kubernetes. Available online: https://kubernetes.io/ (accessed on 3 January 2019).
35. Abdollahi Vayghan, L.; Saied, M.A.; Toeroe, M.; Khendek, F. Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned. In Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, CA, USA, 2–7 July 2018; pp. 970–973.
36. Modak, A.; Chaudhary, S.D.; Paygude, P.S.; Ldate, S.R. Techniques to Secure Data on Cloud: Docker Swarm or Kubernetes? In Proceedings of the 2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT), Coimbatore, India, 20–21 April 2018; pp. 7–12.
37. Docker Swarm. Available online: https://docs.docker.com/engine/swarm/ (accessed on 3 January 2019).
38. A Distributed, Reliable Key-Value Store for the Most Critical Data of a Distributed System. Available online: https://coreos.com/etcd/ (accessed on 11 January 2019).
39. Goel, L.B.; Majumdar, R. Handling mutual exclusion in a distributed application through Zookeeper. In Proceedings of the 2015 International Conference on Advances in Computer Engineering and Applications, Ghaziabad, India, 19–20 March 2015; pp. 457–460.
40. Redis. Available online: https://redis.io/ (accessed on 3 January 2019).
41. Apache NiFi. Available online: https://nifi.apache.org/ (accessed on 3 January 2019).
42. Robinius, M. Strom- und Gasmarktdesign zur Versorgung des Deutschen Strassenverkehrs mit Wasserstoff. Ph.D. Thesis, RWTH Aachen University, Aachen, Germany, 2015. (In German)
43. Robinius, M.; Otto, A.; Syranidis, K.; Ryberg, D.S.; Heuser, P.; Welder, L.; Grube, T.; Markewitz, P.; Tietze, V.; Stolten, D. Linking the power and transport sectors—Part 2: Modelling a sector coupling for Germany. *Energies* **2017**, *10*, 957. [CrossRef]
44. Ryberg, D.S.; Robinius, M.; Stolten, D. Evaluating Land Eligibility Constraints of Renewable Energy Sources in Europe. *Energies* **2018**, *11*, 1246. [CrossRef]

45. Geospatial Land Availability for Energy Systems (GLAES). Available online: https://github.com/FZJ-IEK3-VSA/glaes (accessed on 4 January 2019).
46. Ryberg, D.S.; Caglayan, D.G.; Schmitt, S.; Linßen, J.; Stolten, D.; Robinius, M. The Future of European Onshore Wind Energy Potential: Detailed Distribution and Simulation of Advanced Turbine Designs. *Preprints* **2018**, 2018120196. [CrossRef]
47. Welder, L.; Ryberg, D.S.; Kotzur, L.; Grube, T.; Robinius, M.; Stolten, D. Spatio-temporal optimization of a future energy system for power-to-hydrogen applications in Germany. *Energy* **2018**, *158*. [CrossRef]