# Dominoes: A ”Simulation-as-a-Service” Platform for User-Friendly Interdisciplinary Co-Simulations

**Hannes Restel**
Fraunhofer FOKUS
hannes.restel@fokus.fraunhofer.de

**Stefan Pfennigschmidt**
Fraunhofer FOKUS
stefan.pfennigschmidt@fokus.fraunhofer.de

## ABSTRACT

This work introduces a *Simulation-as-a-Service platform* designed to simplify the creation, management, and execution of simulation components for interdisciplinary co-simulations in urban disaster management. The platform can be used by simulation experts from various domains, such as first-responder coordination, supply grids, and communication networks. The modular and extensible platform supports the SimulationOps process and adapts to diverse and novel use cases and simulation domains. It enables seamless simulation execution across any technology, deployment, or environment, from personal computers to clusters or cloud systems. Users can develop simulation components using their preferred domain-specific technologies, including programming languages and simulation tools, while a platform-specific encapsulation ensures their compatibility. The presented Simulation-as-a-Service platform is implemented using a microservice architecture with an event-driven communication pattern. It has been successfully tested in a laboratory setting to simulate an urban flooding scenario.

### Keywords

Co-Simulations, Digital Twin, Crisis Management, Software Architecture, Microservices

## INTRODUCTION

Assessing the vulnerability of urban areas and their vital infrastructure to natural disasters—such as floods, severe thunderstorms, or heat waves—has become an essential research focus in recent years, as these hazards tend to increase in frequency and intensity due to climate change (Calvin et al., 2023). Resulting failures or disruptions in critical infrastructure can lead to significant consequences, including persistent shortages and serious threats to public safety. SIRIOS (”Security of Socio Technical Systems”) is a joint project aiming to provide a holistic view of these vulnerabilities and their impacts to improve resilience (“SIRIOS — Fraunhofer Center for the Security of Socio-Technical Systems SIRIOS”, 2025).

As part of this project, a *digital twin* (Trauer et al., 2020) of a downtown area of Berlin, Germany, has been developed to simulate the cascading effects of severe weather events under various scenarios, such as flooding caused by heavy rainfall. This digital twin integrates *simulation components* for static elements (e.g., buildings) and dynamic infrastructure components (e.g., water, gas, electricity, communication networks) as well as human actors (e.g., emergency response teams, volunteers). These simulations provide insights for all four phases of the crisis management cycle: mitigation (also referred to as prevention), preparation, response, and recovery (Johansson, 2007).

The interdisciplinary nature of these co-simulations requires collaboration between domain experts of SIRIOS, each contributing simulation components developed using their own domain-specific tools, models, and methodologies. Researchers use these simulations to address objectives such as: (i) identifying critical paths and cascading effects in infrastructure networks, (ii) optimizing the placement of base stations for nomadic networks during blackouts, (iii) rerouting emergency vehicles to avoid flooded areas, and (iv) improving communication strategies to guide citizen behavior during disasters.

Figure 1 depicts an exemplary setup of how the dedicated simulators are combined into a co-simulation network of the Berlin flooding scenario, which we will refer to in this work.

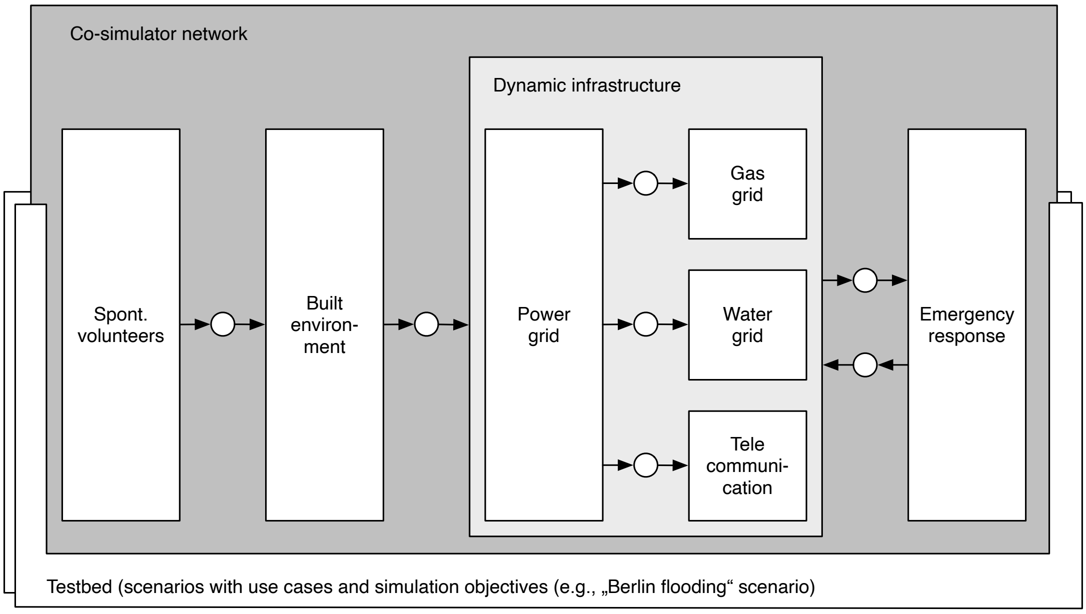


**Figure 1. One example of a Co-Simulation Network of the "Berlin flooding" scenario**

*Simulators* are software components that compute simulation tasks based on a defined input using problem-specific algorithms and models. They are typically developed in different programming languages and simulation tools, and they may have different data formats. In our joint project, several simulators have been created to work on the digital twin, e.g., *built environment, power, drinking water, gas, cellular network, volunteers,* and *emergency response services* (Martini et al., 2024). Depending on the simulation goal, all or just a subset of the simulator components are combined to run a co-simulation. For example, Lukau et al. address emergency communication during blackouts by coupling a *power grid simulator* and a *cellular network simulator* (Lukau et al., 2025).

Previous research introduced the concept of *SimulationOps* (*SimOps*) that proposed a collaboration process to realize interdisciplinary co-simulations in an effective way to get results fast ("value and velocity") (Restel, 2023). *SimOps* merges together the development process of how to devise single simulators and then to combine them, and the operating process of the co-simulations. SimOps combines concepts from Design Science Research (Peffers et al., 2006), DevOps ("DevOps", 2022), "as-a-service" platforms ("Software as a Service", 2025), and Data Mesh (Dehghani, 2022).

In this work, we close the gap and present a suitable *Simulation-as-a-Service platform* as a technical implementation of the *SimOps* process that adopts a user-friendly self-service approach, enabling simulation experts to:

a) integrate their simulation components into a reusable and technology-agnostic platform, and

b) create, manage, and execute co-simulation scenarios within a unified environment.

Simulation experts typically possess deep domain knowledge rather than software development skills. The platform, therefore, is designed to shield them from the technical details of deploying and running simulations, allowing them to focus on their core business activities. User groups of the platform are researchers in an initial phase, which is intended to evolve into a self-service platform for end users, such as city planners or emergency managers.

Conducting simulations should ideally be possible both pre-crisis and ad-hoc during disasters, which is why the platform supports deployment on single nodes (e.g., laptops), local clusters, or distributed cloud environments, making it adaptable to diverse research and application contexts. For example, during sudden-onset disasters, the platform can operate locally without requiring server infrastructure or internet connectivity.

The platform employs a microservices-based architecture that is modular, extensible, and scalable.

An early iteration of the Simulation-as-a-Service platform has been successfully used in a laboratory setting to simulate the aforementioned flooding scenario.

This paper is structured as follows: Section Related Work reviews related work, Section Design Objectives for the Platform outlines the business requirements and design objectives for the platform, Section Co-Simulation Platform presents the platform's components, and Section Using *Dominoes* to create Co-Simulations for the "Berlin Flooding" Scenario describes its technical implementation and user interactions. Finally, Section Conclusion and Outlook concludes the paper and provides an outlook on future work.

## RELATED WORK

Co-simulation is a simulation technique used in simulation modeling to analyze complex systems that cannot be adequately represented by a single model. It involves the integration of multiple simulation tools or models to provide a comprehensive understanding of a system's behavior. Distributed co-simulation further extends this concept by enabling the execution of different simulation models in a distributed computing environment. This approach proves particularly advantageous for either large-scale systems, systems that incorporate models requiring specific hardware, or flexible service-based simulation systems.

Over the past decades, numerous models have been proposed on how to develop and run co-simulations. The most widely recognized concept is the High Level Architecture (HLA), publicly available as a set of standards ("IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Framework and Rules", 2010). This concept describes a federated system of multiple simulators, along with a common runtime infrastructure to facilitate communication and synchronization. One example of a simulation platform that uses the HLA approach in the domain of mobility and communication is Eclipse Mosaic (Schrab et al., 2023).

The CSIAC's approach to "Modeling & Simulation as a Service" (MSaaS) (Robert Siegfried et al., 2018) emphasizes the use of cloud computing and service-oriented architectures to deliver modeling and simulation capabilities effectively. "As-a-service" refers to a self-service approach, where users can access and use services on-demand. This is combined with a business model where users pay for the services they use, e.g., pay-per-use or subscription-based. Gütlein and Djanatliev (2020) present an architecture for MSaaS, which focuses on Apache Kafka for communication between and coordination of components participating in a simulation.

The Interdependent Networked Community Resilience Modeling Environment (IN-CORE) is a computational environment addressing modeling using geospatial models of communities (van de Lindt et al., 2023). The approach is "propagating the effect of physical infrastructure damage and loss of functionality to social and economic impacts" ("IN-CORE", 2025). It is a promising approach related to solving similar problems as our approach on the business side. It includes capabilities for modeling natural disasters (e.g., tsunamis, earthquakes, flooding) and can also include social models. However, it seems to focus on scientific analyses rather than co-simulations and seems to miss relevant simulators for our use cases (e.g., for emergency response or for infrastructure services such as power, gas, drinking water). Workflows can be defined that chain together analysis steps, but it seems that co-simulations cannot be performed, and it is unclear if and how feedback loops can be modeled from one component to another. On the technical platform side, modern industry-standard technologies such as Docker ("Docker", 2025) and Kubernetes ("Production-Grade Container Orchestration", 2025) are used for the service-oriented architecture that is divided into core web services (e.g., for fetching data sets), web tools, and components to create the analysis workflows. The services are centrally hosted. The platform is not programming language agnostic, as only Python is supported. The platform is published as open-source software and welcomes contributions from the worldwide community ("IN-CORE", 2025), yet the process of how to design and contribute for new (simulation) services is not outlined.

To the best of our knowledge, there is no currently available concept that fully supports the requirements identified by the SimOps approach. For instance, the emphasis on a strict separation of the various development and operation responsibilities with respect to co-simulation platforms, or the desired flexibility in forms of deployment.

Domain-Driven Design (DDD) is a concept originally intended for software development (Evans, 2003) to improve the design of complex systems by breaking them into smaller, independent components by creating "*bounded contexts*". Each bounded context represents a distinct business capability with its own domain model, enabling clear separation of concerns and establishing a common language between domain experts and developers. This approach aligns well with interdisciplinary co-simulations, where independent domains (e.g., power grids, emergency response, communication networks) must be integrated into a cohesive system. The tools and techniques of DDD can thus support the development of the Simulation-as-a-Service platform platform for co-simulations. On the technical level, the *Microservices architecture* style (Richards & Ford, 2020) is a suitable candidate for implementing DDD concepts. Microservices structure an application as a collection of loosely coupled, self-contained services, each implementing a specific business capability with its own data models and persistence. Unlike monolithic architectures, microservices avoid a global database and communicate through well-defined APIs, either synchronously (e.g., REST (Fielding, 2000)) or asynchronously (e.g., message brokers). This architecture enhances flexibility, scalability, and resilience, making it easier to adapt to changing requirements and conditions.

## DESIGN OBJECTIVES FOR THE PLATFORM

As a prerequisite for designing and implementing the *Simulation-as-a-Service platform*, this section first describes the design objectives, then derives a concrete process (”user journey”) for realizing co-simulations on a self-service platform, and evaluates potential architecture styles for the platform. The design objectives were directly gathered from the stakeholders/researchers of our research community, and thus represent the business requirements.

The simulation platform must:

- support the *SimulationOps* process.
- support different simulation domains.
- be simple to use for simulation experts of different domains, that often are not software developers.
- allow simulation components to be developed by the domain simulation experts with their technologies of choice, e.g., tools, programming languages, algorithms, and data formats.
- allow for rapid testing and validation of the feasibility and technology maturity of the simulation components.
- allow users to reuse and plug together existing components on the platform in a self-service manner.
- allow users to run single simulations as well as complex co-simulation scenarios, and to evaluate the results.
- support the execution of such scenarios on local machines, cloud-based infrastructures, or hybrid environments.
- be modular and extensible to support co-simulations across the different domains, that are plugged together in unpredictable ways depending on the use case and simulation objective.

### Identifying the Appropriate Process

The existing concept of *SimOps* proposes a collaboration process between three distinct roles that develop and use simulation components, so that interdisciplinary co-simulations can be realized in a user-friendly and effective way (Restel, 2023):

- *Producer Role:* Creates simulation components within clearly defined bounded contexts, such as simulators, co-simulators, and data products.
- *Consumer Role:* Uses simulation components to run simulations, gaining knowledge and support for their hypotheses and business goals.
- *Supporter Role:* Supports producers and consumers by reducing the complexity of the designed components through adequate abstractions. Also provides infrastructure for the platform, enabling producers and consumers to use the platform effectively.

*SimulationOps* realizes a *DevOps* process (“DevOps”, 2022) allowing the dedicated development and operations teams to work together efficiently by mutual feedback and automation for software delivery, infrastructure changes, and operations.

In Figure 2, the main use cases of the intended *Simulation-as-a-Service platform* are shown, and the responsible roles from *SimOps* are assigned to the use cases. The use cases are: (i) create simulation components, (ii) knit together and configure components for co-simulation scenarios, (iii) execute/run a co-simulation, (iv) evaluate the results of a simulation run, (v) provide guidance for Producers/Consumers, and (vi) operate the platform.

Co-simulations combine basic simulation components as their building blocks, thus multiple SimOps processes interlock: first to create basic components, then to combine them into co-simulation scenarios. This also means the roles of consumer and producer might overlap, as a producer of a (complex) component may consume basic components.

Thus, we extend the three generic *SimOps* roles (*Consumer, Producer, Supporter*) and derive more specialized roles for the platform, to better reflect the composition steps from basic/single components towards more complex simulation scenarios facilitating the ”self-service” pattern. The refined roles are:

- *Domain experts* produce basic simulation components. Derived from *Producer*.
- *Co-simulation experts* create co-simulation scenarios from existing simulation components. Derived from *Consumer, Producer*.
- *Simulation users* configure existing co-simulation scenarios and run the simulations. Derived from *Consumer*.
- *Platform developers* support domain experts and co-simulation experts in creating and managing simulation components and scenarios. They make the components compatible with the platform. Derived from *Supporter*.
- *Operators* operate and manage the platform infrastructure and support in managing all the simulation artifacts. Derived from *Supporter*.

Figure 3 shows the interlocking SimOps processes for the extended roles and visualizes how a role can be a consumer as well as a producer of simulation components.

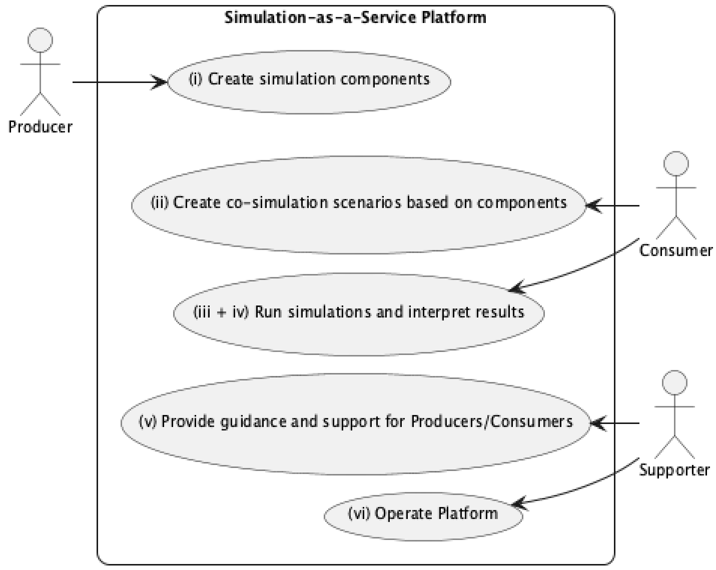


**Figure 2. Use Cases for the Simulation-as-a-Service platform as realized with the roles defined in SimOps**

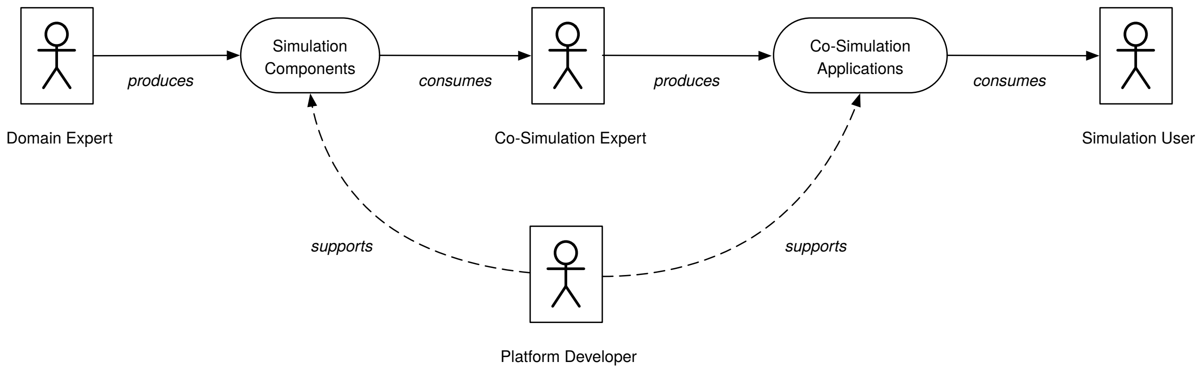


**Figure 3. Interlocking SimOps processes for the extended roles**

### Identifying the Appropriate Software Architecture Style

Choosing an appropriate software architecture style is a crucial step in the design for the *Simulation-as-a-Service platform* as it defines important characteristics of the system, such as the performance, and maintainability, but also the "fitness" of the system to solve the problems at hand. A vast variety of potential software architecture styles exist with very different properties.

To identify a fitting software architecture for a software system Richards and Ford suggest to first identify and prioritize the *architecture characteristics* that are important for the system, and then to choose the architecture style that fulfills the chosen characteristics best (Mark Richards, 2025; Richards & Ford, 2020). We followed this methodology and mapped the business requirements to the full set of *architecture characteristics*, and then identified the top 5 driving characteristics for the *Simulation-as-a-Service platform*, which are: (i) *Adaptability* (Readiness to accommodate changes), (ii) *Interoperability* (Interaction with other systems), (iii) *Deployability* (Ease and speed of deployment), (iv) *Abstraction* (Level to which parts are isolated from other parts), (v) *Configurability* (Customization to meet specific needs).

As the next step of the methodology, we rated the architecture characteristics for each architecture style. The rating is based on the template provided by Mark Richards (Mark Richards, 2025) that considers the most widely used eight architecture styles (*Layered Architecture*, *Modular Monolith*, *Microkernel Architecture*, *Microservices Architecture*, *Service-Based Architecture*, *Service-Oriented Architecture (SOA)*, *Event-Driven Architecture (EDA)*, and *Space-Based Architecture*). Each architecture characteristic is supported by a certain fit from 1 ("not a good fit") to 5 ("perfect fit") for each architecture style. For the identified five driving characteristics, we summed up the 5-star ratings for each architecture style in Table 1. The architecture styles with the highest total points are considered the best fit for the platform.

**Table 1. Architecture styles candidates for the platform**

| **Architecture Style** | *Abstraction* | *Adaptability*[1] | *Deployability*[2] | *Interoperability* | Cost | **Points** |
|---|---|---|---|---|---|---|
| **Layered (†)** | 1 | 1 | 1 | 1 | $ | 4 |
| **Modulith (‡)** | 1 | 1 | 2 | 1 | $ | 5 |
| **Microkernel (‡)** | 3 | 3 | 3 | 3 | $ | 12 |
| **Microservices (‡)** | 1 | 5 | 5 | 3 | $$$$$ | 14 |
| **Service-Based (‡)** | 1 | 4 | 4 | 2 | $$ | 11 |
| **Service-Oriented (†)** | 5 | 1 | 1 | 5 | $$$$ | 12 |
| **Event-Driven (†)** | 4 | 5 | 3 | 3 | $$$ | 15 |
| **Space-Based (†)** | 1 | 3 | 3 | 2 | $$$$ | 9 |

**This table rates the architecture styles that are considered for the platform, along with the ratings for the driving characteristics and the total points. The architecture styles are rated on a scale from 1 to 5, with 5 being the best fit. The cost is rated on a scale from 1$ to 5$, with 1 being the lowest cost. The total points are the sum of the ratings for the driving characteristics. *Configurability* has been merged with *Deployability*. Symbols: † - Technical Partitioning, ‡ - Domain Partitioning.**

The EDA leads with 15 points, followed by Microservices with 14 points, and SOA and Microkernel Architectures with 12 points each.
First considering the *domain partitioning*, i.e., the properties of an architecture to support dividing a system into distinct functional areas or domains each responsible for a specific aspect of the system's overall functionality, the Microkernel architecture cannot be used, as it is not a distributed architecture, leaving the *Microservices architecture* as the best fit for the platform on the domain level during runtime.
Secondly, considering the *technical partitioning*, e.g., how components will communicate with each other, both *EDA* and *SOA* seem to be a good fit for the platform. For those two options, EDA is preferred over SOA for our use case, as EDA is particularly well-suited to be integrated with Microservices, supports asynchronous communication patterns, and is easy to set up and to maintain also on single-laptop deployments.

The Microservices Architecture is rated with the high cost as it requires more effort to set up and maintain the architecture yet the cost is not a driving characteristic.
Microservices with an event-driven communication pattern allow for a very good decoupling of the simulation components, and thus allow the domain experts and co-simulation experts to develop their own components with their technologies of choice. Plugging the simulation components together into co-simulations can be done on the surrounding layer of the platform, only based on a thin interface that is easy to integrate.

## CO-SIMULATION PLATFORM

After presenting the requirements, processes, and roles, as well as the preferred software architecture style, this section describes the designed *Simulation-as-a-Service platform* itself.

Designing the platform requires partitioning the essential functionalities along the roles and responsibilities identified by the SimOps process. Subsequently, the components need to be mapped to the concept of event-driven microservices. Integrating existing simulator systems, each with its own user interfaces, simulation control, and communication facilities, presents a significant challenge. To achieve complete separation of application and platform functionality, the following considerations must be taken into account:

- *Control flow*: simulators are required to surrender, for instance, time control facilities so that they can be controlled from the outside.
- *Data flow*: simulators are to relinquish control of data exchange to facilitate their seamless integration into flexible data processing pipelines.
- *Communication technology*: a simulator's algorithmic component needs to be separated from any integrated communication mechanisms to facilitate its deployment in diverse runtime environments.
- *Deployment specifics*: a simulator's functionality needs to enable cross-platform execution, allowing it to run on various environments, including local machines, cloud services, and hybrid configurations.

[1] Adaptability is mapped to Evolvability, as only Evolvability is available in the template by Richards.
[2] Configurability has been integrated and merged together with Deployability, as the template by Richards does not provide configurability. Based on our experience, deployability is the next best match, and configurability could also be considered as a sub-characteristic of deployability.

### Platform Components

The platform comprises four major types of components: (i) Processing units, (ii) Services, (iii) Applications, and (iv) Orchestrator(s).

*Processing units* implement simulation algorithms or other data processing tasks as passive and independent building blocks providing a common application programming interface (API) to facilitate controlling their execution from the outside. Processing units are, in general, the responsibility of *domain experts* and can be written in multiple programming languages.

*Services* encapsulate processing units and provide life cycle management and inter-process communication interfaces. They offer mechanisms for creating unit instances, initializing them for a simulation run, and invoking their functions to apply external data, execute subsequent steps, or query their state. This enables running multiple simulation algorithms in a coordinated manner in distributed environments and allows processing units to participate in multiple concurrent simulations for various users, objectives, and parameter sets. The provisioning of service capsules is the responsibility of the *platform developers*.

*Applications* describe co-simulation scenarios. They specify the required simulation services, their dependencies, initial state configurations, and simulation parameters, including model time, simulation duration, step widths, and other relevant settings. Applications are, in general, defined by *co-simulation experts*.

*Orchestrators* accept application descriptions and initiate and execute corresponding simulation runs. Consequently, an orchestrator establishes connections to the requisite services and invokes them sequentially (until the application's termination condition is met), while ensuring that data necessary for a service to execute a specific step is retrieved and provided in a timely manner. Orchestrators are an immediate component of the platform, so the *platform developers* are responsible for their development.

#### *Processing Units*

Processing units are designed to implement simulation algorithms as self-contained, passive components that are independent of a specific execution environment. At its core, a unit is essentially a function $f(state_{ti-k}, t_i) \mapsto state_{ti}$ (step function) that takes a model state and a corresponding point in model time and produces a subsequent state. There is a well-defined interface for initializing and retrieving model data and for applying external events to a model state, enabling the units to be used in a cooperative environment. A processing unit instance manages its own local model independently of any other unit. It may use external resources such as databases to store and retrieve its model data. Units may also use shared *read-only* resources such as lookup services. Units can be written in any programming language that has a language binding to the platform; currently, this includes JavaScript, Python, Java, Kotlin, and C#.

A processing unit implements the following API.

- `init (options)`: initializes the state of the unit with the given options.
- `apply (time, data)`: applies the given external data to the state of this unit for the given point in model time.
- `step (time)`: computes state changes for the given point in model time.
- `query (time, filter) : data`: queries the state for the given point in model time using a unit-specific query/filter expression.
- `release ()`: releases all resources held by this unit.

An API contract exists that stipulates additional conditions under which a unit's functions are executed.

Depending on data flow patterns, processing units can be classified into three types.

- *Generators* produce data; however, they do not accept external events. Examples include clocks, replay, interpolator, or control units.
- *Collectors* accept external events; however, they do not produce any data. Examples include consoles, archives, displays, or monitors.
- *Transformers* do both: they accept external events and produce data. Examples include simulators, filters, and data converters.

Thus, from a business point of view, *Transformers* are the most frequent—and potentially most important—components because the simulators themselves fall into this category.

### *Services*

Processing units are pieces of software (similar to library modules) that require an executable frame. By design, units are passive, lacking the capability to initiate communication independently. For units to participate in a simulation, the platform offers service capsules. These capsules provide units with the necessary runtime environment and interprocess communication facilities to enable data exchange, as well as orchestrators with interfaces to control data flow and service execution.
Depending on desired architectural properties, service capsules can be implemented based on various technologies or architectural styles. The chosen style, communication pattern, or technology employed by a service implementation determines their platform "flavor". Services can, for instance, provide communication and control interfaces based on REST, gRPC, message queues, or data streams. To accommodate processing units written in diverse programming languages, service implementations provide the necessary bindings for each flavor.
Integrating a processing unit with a service capsule results in an executable service, and while a processing unit instance is dedicated to a single simulation, a service instance can concurrently support multiple simulations for various users, objectives, and parameter sets.

### *Orchestrators*

Orchestrators are responsible for executing simulation runs by sequentially invoking the necessary services and ensuring that data required for a service to execute a specific step is retrieved and provided in a timely manner. In a typical co-simulation scenario, services are interdependent, i.e., the output of one service serves as the input of another. Figure 4 illustrates this principle using an example of four interdependent services collectively executing the steps of a simulation run in the form of a block diagram based on Fundamental Modeling Concepts (FMC) (Knöpfel et al., 2006).

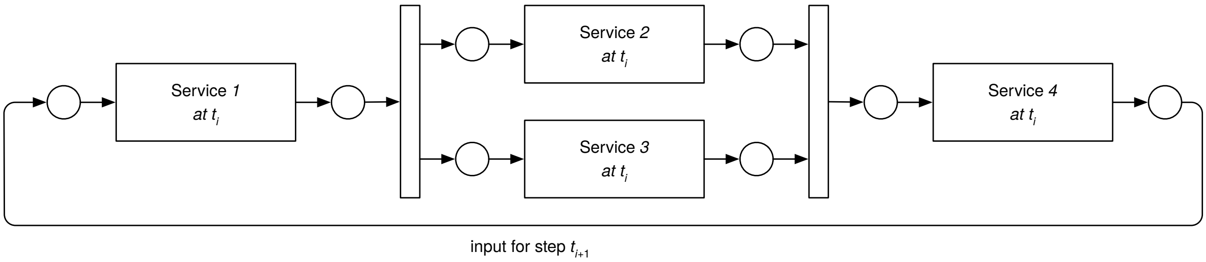


**Figure 4. FMC diagram of service dependencies in a co-simulation environment. The labeled boxes represent the state of services at a time $t$; information flows along the arrows through channels (represented as circles); the horizontal bars represent synchronization points, to illustrate that a service waits for all input to be available before proceeding with its own next step.**

Consequently, for every relevant time point in a simulation, the orchestrator must guarantee that the services execute in the correct order and that the data produced by one service is available to the subsequent service(s) in the chain. Orchestrators provide an interface for initiating and controlling simulation runs by accepting and interpreting such simulation scenarios (or applications). Similarly to services, an orchestrator is associated with a specific platform flavor and supports its communication protocol and technology to enable it to manage its services effectively.

### *Applications*

Simulation scenarios are described by applications. They specify the required simulation services, their dependencies, initial state configurations, and simulation parameters, including model time, simulation duration, step widths, and other relevant settings (Listing 1). Applications can be executed by the simulator service network through the use of an orchestrator.

**Listing 1. Example of an application definition.**

```
{
    "app": {
        "name": "example-app",
        "environment": { "start": 0, "end": 3600 },
        "nodes": [
            { "name": "simulator-1", "options": { ... } },
            { "name": "converter", "options": { ... } },
            { "name": "clock-control", "options": { ... } },
```

```
            { "name": "display", "options": { ... } }
        ],
        "dependencies": [
            { "source": "simulator-1", "sink": "converter" },
            { "source": "converter", "sink": "simulator-2" },
            { "source": "simulator-2", "sink": "display", "filter": "..." },
            { "source": "clock-control", "sink": "simulator-1" }
        ]
    }
}
```

Figure 5 depicts an example simulator network that communicates with an external user interface system. The buttons of the user interface (UI) are used to pause or resume the simulation through a clock control unit, which subsequently generates clock events. Simulator 1 depends on the clock ticks and the results of simulator 3; simulators 2 and 3, in turn, depend on the results of simulator 1. An additional converter unit is employed to convert data between simulator units 1 and 2. The simulation results of units 2 and 3 are collected to be displayed in the UI.

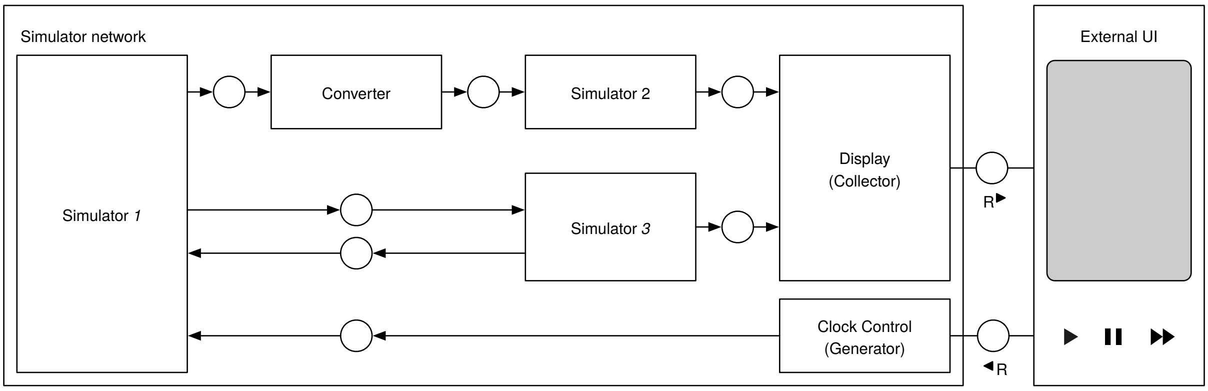


**Figure 5. Schematic FMC representation of units interacting in and outside a simulator network.**

## *Dominoes* as a Platform Flavor

The previous section introduced the components of the platform and their interactions at a conceptual level. A practical realization of such a platform can be achieved through numerous architectural styles and technology choices. In this section, we present "*Dominoes*," a concrete implementation (or "flavor") of the platform that employs a specific architectural style and technology stack, enabling the components to actually run on an IT infrastructure.

*Dominoes* adopts an architecture based on loosely coupled microservices that communicate in an event-driven manner via named finite streams, as previously outlined in Section Design Objectives for the Platform.
*Dominoes* comprises a *static artifact layer* and a *dynamic runtime layer*, as depicted in Figure 6. The *static artifact layer* holds the reusable simulation components (base units, processing units, applications) stored in the *Component Repository*. The *dynamic runtime layer* manages the "running" simulations and the data that simulations consume and produce. Simulations are triggered by the users in selecting which application to run, including a configuration. The platform starts a simulation by spinning up the necessary orchestrator and services and connecting them according to the application description. As we use the *Dominoes flavor* of the platform, the runtime layer uses asynchronous communication between the different components and orders the execution steps for the simulation using *finite streams*.

Using the *Docker* container virtualization technology enables *Dominoes* to separate the defined static artifacts (including all their dependencies, including the operating system) from the underlying infrastructure ("Docker", 2025), so that the static artifacts can be designed in a wide range of programming languages and frameworks. This architecture also enables the deployment and utilization of services in diverse environments, ranging from local machines to cloud infrastructures, regardless of the available hardware or the necessity of installed third-party services.

Each static simulation component of the *Component Repository* is executed as a *dynamic artifact* upon running a simulation and thus becomes a running *service*. In order for a processing unit to be run as a service, it first needs to

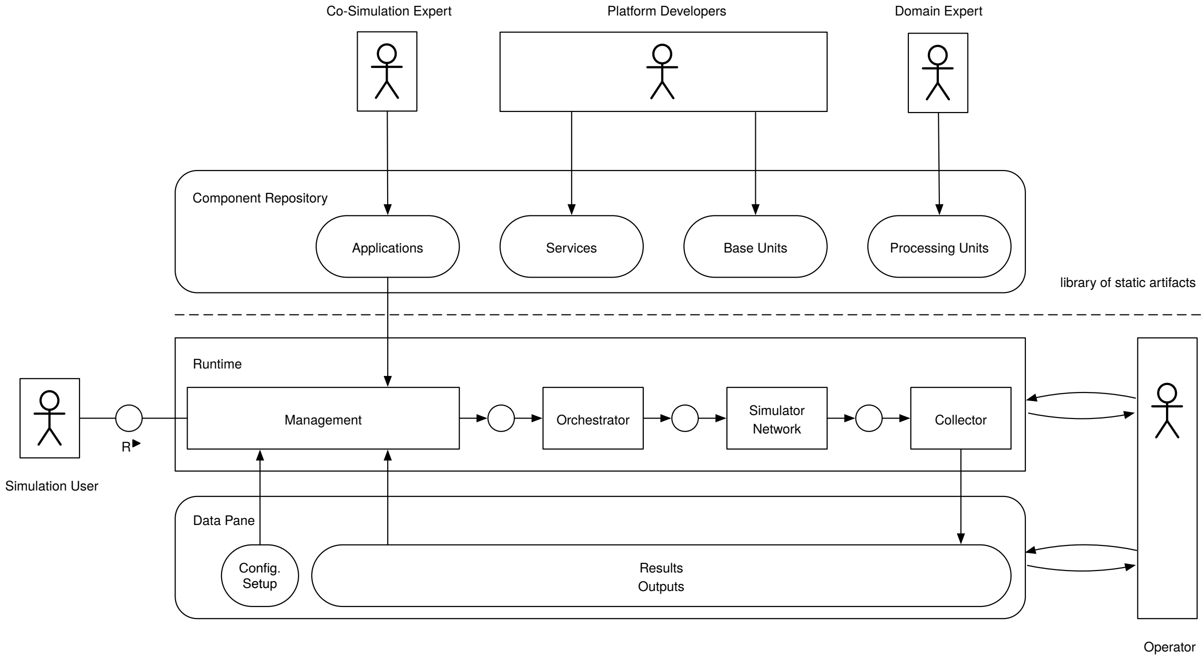


**Figure 6. Static and Dynamic Layers in Dominoes**

be encapsulated in a *Docker image* that provides the technical binding for the runtime. As the encapsulation is identical for each base unit and processing unit, a template is used for creating the Docker images. The *Platform Developers* provide such templates as Docker base images, including the available programming language bindings (e.g., Node.js, Python, C#, Kotlin, or Java) as well as the technical communication bindings of *Dominoes*. The processing units are then simply "dropped" into a base image, and the service image is built. These service images can then be used locally, for instance, through a Docker Compose configuration or by integrating them into a server/cloud deployment.

The platform layers are complemented by predefined orchestrators and a set of generically utilizable fundamental *base units*. Base units encompass, for instance, configurable clocks; collectors that record data to consoles, files, or databases; data suppliers that can be connected, for example, via REST or message queues; a replay unit that can retrieve events from a configurable stream and inject them into a simulation run; a generic data interpolator; and a transformer, which can be configured by specifying transformation expressions using JSON data manipulation languages, such as *JSONata* (JSONata, 2025) or *apath4* (a-f-m, 2025).

Additionally, when run on a server cluster, the platform offers a container management solution that provides an auto-scaling service infrastructure based on Kubernetes ("Production-Grade Container Orchestration", 2025). This container management can also be utilized for automated unit tests or to support integration and end-to-end testing of the services developed.

*Dominoes* prescribes the implementation of two distinct service interfaces, enabling the running dynamic artifacts to be controlled and to exchange data with one another: (i) the *Command Interface*, which enables the orchestrator to assign tasks to services necessary for executing a given simulation scenario (commands define the type of task to be performed and, where applicable, specify the data streams to be used during task execution); and (ii) the *Data Interface*, which facilitates the exchange of data between services during the execution of assigned tasks.
As shown in Figure 7, upon receiving an application scenario, a *Dominoes* orchestrator first generates initialization commands for all the services that are to participate in the simulation. These commands are sent to the services over a command queue. Subsequently, the orchestrator commences generating step invocations for the services. Each step invocation specifies dedicated input and output streams. Upon receiving a step invocation, a service establishes a connection to the specified input stream and awaits data. A service can register several subsequent steps concurrently. The computation of state changes for a step commences only upon receipt of the complete signal from the input stream, indicating that all data from upstream services has been received. Subsequently, the state is queried, and the resulting data is published to the output streams specified for the step. Finally, the orchestrator issues finish commands for all participating services.

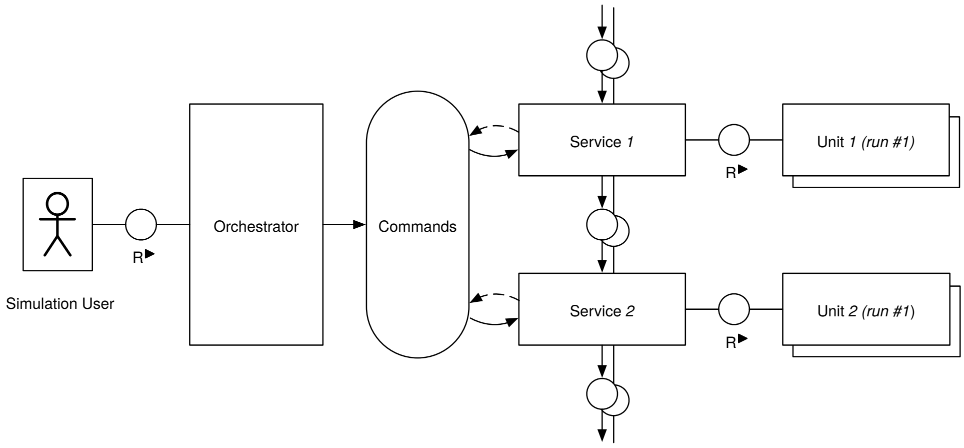


**Figure 7. Command and data interfaces of the Dominoes flavor of the *Simulation-as-a-Service platform*. On the horizontal axis, exchanged information on the command interface is shown, while the vertical axis shows the data exchanged on the data interface.**

## USING *DOMINOES* TO CREATE CO-SIMULATIONS FOR THE "BERLIN FLOODING" SCENARIO

In this section, we present how the SimOps process and roles are integrated with the *Dominoes* platform flavor. It demonstrates how our researchers created, managed, and executed interdisciplinary co-simulations in our testbed of the "Berlin flooding" scenario using a first iteration of *Dominoes*.

In our testbed, the platform has been used for initial laboratory-grade tests employing models designed to simulate the characteristics of diverse infrastructure networks, encompassing gas, power, drinking water supply, and telecommunications systems, to assess the vulnerabilities of buildings in the event of flooding, as well as an agent-based model that facilitates the simulation of emergency response scenarios. Six interconnected processing units to create the complex co-simulation scenario of the flooding have been used, each using different programming languages (Python, Kotlin, and C#) and a variety of data sources and data sinks for the data pane (files, PostGIS database, Redis, FROST, SensorThings API).

The following process showcases the performed steps from inception to execution of our simulation scenario reflecting our instantiation of a SimOps process. The process consists of three phases: In the *setup phase* (steps 1–4), the platform and tooling itself was installed on the workstations and server environment. In the *creation phase* (steps 5–9), the simulation components were created, tested, and uploaded to the platform to await execution. Finally, in the *simulation phase* (steps 10–13), the co-simulation scenario is initialized with input data, executed as a simulation, and evaluated. The process is depicted in the following sequence.

1. The stakeholders agree to use the *Dominoes flavor (d6s)* of the platform.
2. *Domain Experts* and *Co-simulation Experts* have set up Docker on their local machines.
3. *Operators* have set up a Kubernetes-based computer cluster that the platform runs on.
4. *Platform Developers* have made available different *base units* and *orchestrators* generically required for each simulation (e.g., clock, console collector, MongoDB database collector). The base components are uploaded to the *Component Repository* of the platform, which is a Docker registry in the case of *Dominoes*. The artifact from the registry can be used/downloaded by all roles.
5. *Domain Experts* from different simulation contexts/domains create new *processing units*. They reuse their existing algorithms/solutions and extend them so that they implement the *processing unit API* (the functions `init, apply, step, query, release`). As it is the first time that the platform is used within the research project, the *Platform Developers* provide guidance and support. Domain experts are not aware of any technicalities further than that API.
6. *Domain Experts* test and validate their single processing units (and later services) in multiple iterations in a single-node environment (i.e., their laptops). A full *Dominoes runtime* including the base units is installed on their laptops. They furthermore add documentation on how to use the components and what feasible configurations are.
7. *Platform Developers* create matching *services* ("capsules") from the processing units in the matching Dominoes flavor by using and minimally adopting provided Dockerfiles. The services are made available to the users by uploading them to the Docker registry.

8. The *Co-simulation Experts* (which may be the same persons as the *Domain Experts*) create a new *application scenario* with the simulation models and parameters. The application configures a subset of the available *processing units* and *base units*. As it is the first time the new processing units are combined, new *transformers* need to be defined between them. Again, the *Platform Developers* provide guidance and support.
9. The *Co-simulation Experts* test their application locally on their laptops using a single-node deployment of *Dominoes*. The application is then uploaded to the platform.
10. The *Simulation User* (which may be the same persons as the *Co-simulation Experts*) accesses the platform, selects the application scenario, and configures it with initial parameters as well as needed models (e.g., a city model with buildings and the local power grid). The models are either already present as "static data" in the *data pane* of the platform or are provided by the user.
11. The *Simulation User* starts the simulation. The platform creates a new *service instance* for the application scenario and uses the matching *orchestrator* to execute the application scenario.
12. The simulation runs for a certain amount of time and writes the results in a defined output format to a console/terminal as well as a target database in the platform's *data pane*. Using the defined interfaces of the platform, the orchestrator informs the *Simulation User* about the progress and the results of the simulation.
13. The *Simulation User* interprets the results. Optionally, the scenario or input data is tweaked, and the steps 10–13 are repeated.

The building blocks created in steps 1–9 (*processing units*, *services*, and *application scenarios*) are uploaded to the *Component Repository*, which is part of the *static artifact layer*. Only in the subsequent steps, those static artifacts are instantiated to run the simulation on the *dynamic artifact layer* triggered by the *Simulation User*. Figure 8 shows the *dynamic layer* for our Berlin flooding scenario. It is a direct extension of Figure 1 from Section Related Work that shows how the seven simulators have been "brought to life" using the *Dominoes* platform.
All simulators are encapsulated in a *service* (i.e., a Docker container) as indicated by the blue boxes. Each simulator may be realized using distinct programming languages and technologies. The *Dominoes*-specific parts are indicated in orange, which are the *Orchestrator* setting up the actual runtime network and the finite streams to send commands and data between the components. The *Base Units* were omitted for reducing the complexity of the figure and can be understood to be the circles in the orange communication arrows.

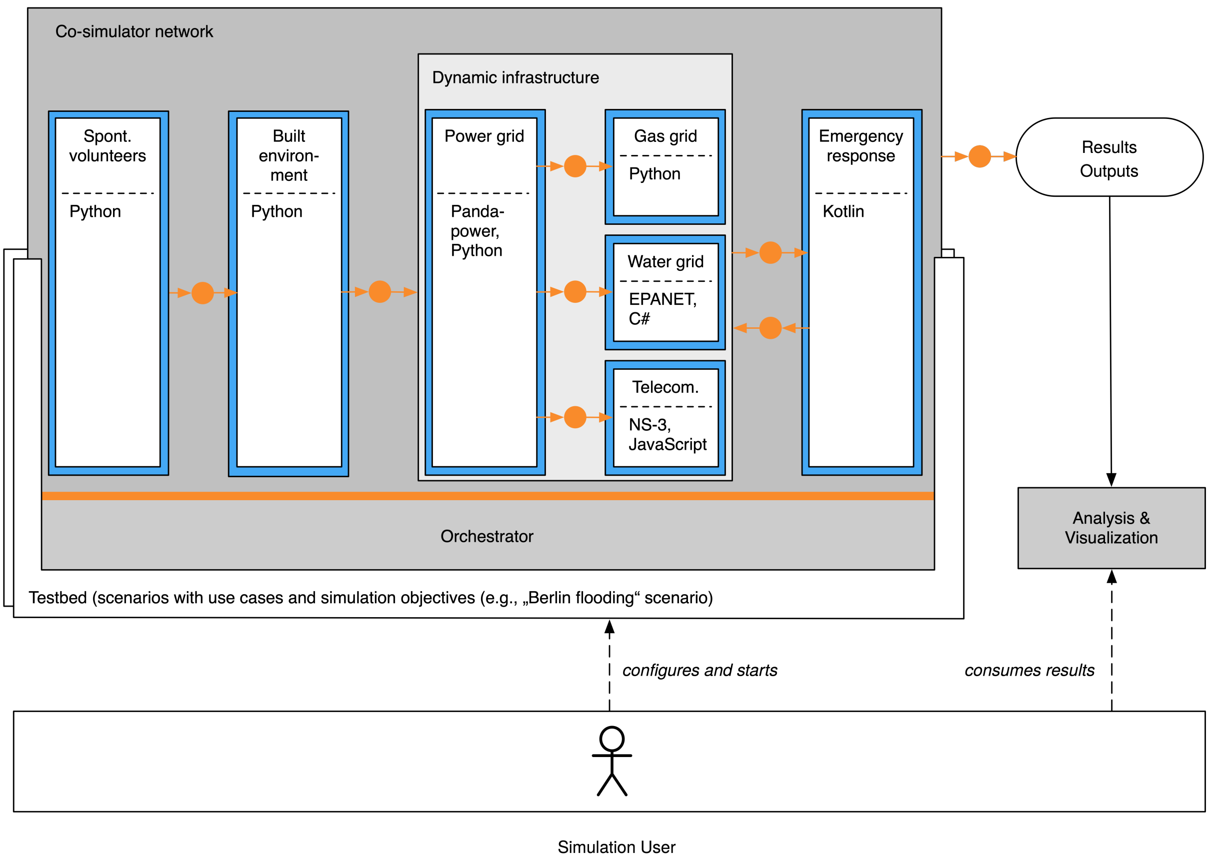


**Figure 8. Static and Dynamic Layers in Dominoes**

## CONCLUSION AND OUTLOOK

In this paper, we presented the design and technical implementation of a Simulation-as-a-Service platform, which we refer to as the *Dominoes Platform*. The platform is designed to support effective and efficient execution of interdisciplinary co-simulations, employing a *SimOps* process. The platform provides capabilities for the creation, testing, and provisioning of individual simulation components, as well as the creation, testing, and provisioning of complex co-simulation scenarios, while ensuring the separation of responsibilities among participating roles. Realized using the modular and extensible microservice architecture style, the platform allows simulation experts to create novel simulation components (processing units, services) and combine them into applications to run interdisciplinary co-simulation scenarios. The key benefit of the platform is that it allows simulation experts to focus on their domain-specific functionality and models, while the platform handles all technical aspects of running the simulations. Existing simulation components can be easily adapted. Encapsulation within a thin platform-provided technical layer facilitates a component's integration into the platform and enables its utilization in a co-simulation. Technologies used in the *Dominoes Platform* to offer this functionality include virtualization (Docker, Kubernetes), as well as event-driven asynchronous communication (message broker, streams). The platform currently supports various programming languages (e.g., Python, JavaScript, Java). Simulation components can be shared through an artifact repository, and can be reused by other simulation experts.

The solution is designed to be very flexible regarding its deployment, so that it can be installed and run on single nodes (such as PCs or laptops), but also on local computer clusters and distributed (cloud) environments.

We have implemented an initial version of the platform, which has undergone laboratory testing to establish its functionality towards the concrete scenario "Berlin Flooding" on our digital twin incorporating seven simulation components.

Future work is intended to focus on the following aspects:

- *Real-world Use Cases:* Demonstrate the platform's functionalities in further testbeds for a range of scenarios.
- *Real-world Processes:* Enable multiple producers to create units and services, and consumers to construct application scenarios utilizing these artifacts to be executed on a cloud-based Kubernetes cluster.
- *Performance Tests:* Test platform components and co-simulation applications to benchmark execution in different environments, such as laptop versus Kubernetes cluster.
- *Data Mesh Support:* Incorporate support for *Data Mesh* concepts, an additional aspect of the SimOps approach that has not yet been addressed by the platform.

## REFERENCES

a-f-m. (2025). *Apath4-workbench*. apath4-workbench. Retrieved February 12, 2025, from https://github.com/a-f-m/apath4-workbench/tree/main/workbench

Calvin, K., Dasgupta, D., Krinner, G., Mukherji, A., Thorne, P. W., Trisos, C., Romero, J., Aldunce, P., Barrett, K., Blanco, G., Cheung, W. W., Connors, S., Denton, F., Diongue-Niang, A., Dodman, D., Garschagen, M., Geden, O., Hayward, B., Jones, C., . . . Péan, C. (2023, July 25). *IPCC, 2023: Climate Change 2023: Synthesis Report. Contribution of Working Groups I, II and III to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change [Core Writing Team, H. Lee and J. Romero (eds.)]. IPCC, Geneva, Switzerland.* (First). Intergovernmental Panel on Climate Change (IPCC). https://doi.org/10.59327/IPCC/AR6-9789291691647

Dehghani, Z. (2022). *Data mesh: Delivering data-driven value at scale* (First edition). O'Reilly.

DevOps. (2022, October 18). In *Wikipedia*.
Page Version ID: 1116813652.

*Docker: Accelerated Container Application Development*. (2025). Retrieved February 21, 2025, from https://www.docker.com/

Evans, E. (2003). *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*.

Gütlein, M., & Djanatliev, A. (2020). Modeling and Simulation as a Service using Apache Kafka. *Proceedings of the 10th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, 171–180. https://doi.org/10.5220/0009780501710180

IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Framework and Rules. (2010). *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, 1–38. https://doi.org/10.1109/IEEESTD.2010.5553440

*IN-CORE*. (2025). Retrieved April 5, 2025, from https://github.com/IN-CORE

Johansson, J. (2007). *Risk and Vulnerability Analysis of Large-Scale Technical Infrastructures* [Licentiate Thesis]. Lund University.

JSONata. (2025). *JSONata Documentation*. JSONata Documentation. Retrieved February 12, 2025, from http://docs.jsonata.org/

Knöpfel, A., Gröne, B., & Tabeling, P. (2006, March). *Fundamental Modeling Concepts: Effective Communication of IT Systems*. Wiley.

Lukau, E., Schiller, J., & Meissen, U. (2025). Simulation as a Tool: Towards Next Generation Crisis Management and Emergency Communication during Blackout Disasters. *Proceedings of the 22th International ISCRAM Conference*.

Mark Richards. (2025). *Developer to Architect*. Developer to Architect. Retrieved January 30, 2025, from https://www.developertoarchitect.com/resources.html

Martini, T., Rosin, J., Vetter, J., Neuhäuser, S., Lukau, E., Catal, F., Boigk, M., Simon, M., Monteforte, M., Gerold, M., Phung, W. F., Dietze, S., Finger, J., Brausewetter, P., & Nicolai, S. (2024). Towards a modular co-simulation framework for the assessment of cascading effects among critical infrastructures and the impact on citizens, 110–120.

Peffers, K., Tuunanen, T., Gengler, C., & Rossi, M. (2006). The design science research process: A model for producing and presenting information systems research. *Proceedings Design Research Information Systems and Technology DESRIST'06*, *24*.

*Production-Grade Container Orchestration*. (2025). Kubernetes. Retrieved February 21, 2025, from https://kubernetes.io/

Restel, H. (2023). SimulationOps – Towards a Simulation as-a-Service Platform for Resilient Societies Using a Cross-domain Data Mesh. In J. Radianti, I. Dokas, N. Lalone, D. Khazanchi, & H. Baharmand (Eds.), *Proceedings of the 20th International ISCRAM Conference* (1st ed., pp. 575–585). University of Nebraska at Omaha. https://doi.org/10.59297/NUER9877

Richards, M., & Ford, N. (2020). *Fundamentals of Software Architecture: An Engineering Approach* (First edition). O'Reilly Media, Inc.
OCLC: on1089438191.

Robert Siegfried, Jon Lloyd, & Tom Van Den Berg. (2018, November 1). *A New Reality: Modelling & Simulation as a Service - CSIAC*. CSIAC Cybersecurity & Information Systems Information Analysis Center. Retrieved February 18, 2025, from https://csiac.dtic.mil/articles/a-new-reality-modelling-simulation-as-a-service/
Journals / Fall 2018: Volume 6 Issue 3.

Schrab, K., Neubauer, M., Protzmann, R., Radusch, I., Manganiaris, S., Lytrivis, P., & Amditis, A. J. (2023). Modeling an its management solution for mixed highway traffic with eclipse mosaic. *IEEE Transactions on Intelligent Transportation Systems*, *24*(6), 6575–6585. https://doi.org/10.1109/TITS.2022.3204174

*SIRIOS — Fraunhofer Center for the Security of Socio-Technical Systems SIRIOS*. (2025). Retrieved January 27, 2025, from https://www.sirios.fraunhofer.com/sirios/en

Software as a service. (2025, January 1). In *Wikipedia*.
Page Version ID: 1266523522.

Trauer, J., Schweigert-Recksiek, S., Engel, C., Spreitzer, K., & Zimmermann, M. (2020). WHAT IS A DIGITAL TWIN? – DEFINITIONS AND INSIGHTS FROM AN INDUSTRIAL CASE STUDY IN TECHNICAL PRODUCT DEVELOPMENT. *Proceedings of the Design Society: DESIGN Conference*, *1*, 757–766. https://doi.org/10.1017/dsd.2020.15

van de Lindt, J. W., Kruse, J., Cox, D. T., Gardoni, P., Lee, J. S., Padgett, J., McAllister, T. P., Barbosa, A., Cutler, H., Van Zandt, S., Rosenheim, N., Navarro, C. M., Sutley, E., & Hamideh, S. (2023). The interdependent networked community resilience modeling environment (IN-CORE). *Resilient Cities and Structures*, *2*(2), 57–66. https://doi.org/10.1016/j.rcns.2023.07.004