Digital Twin Development with OpenTwin
Architecture and Extensibility

Alexander Küster, Jan Wagner, Peter Thoma
Frankfurt University of Applied Sciences, Faculty of Computer Science and Engineering, Frankfurt am Main, Germany

What is OpenTwin?

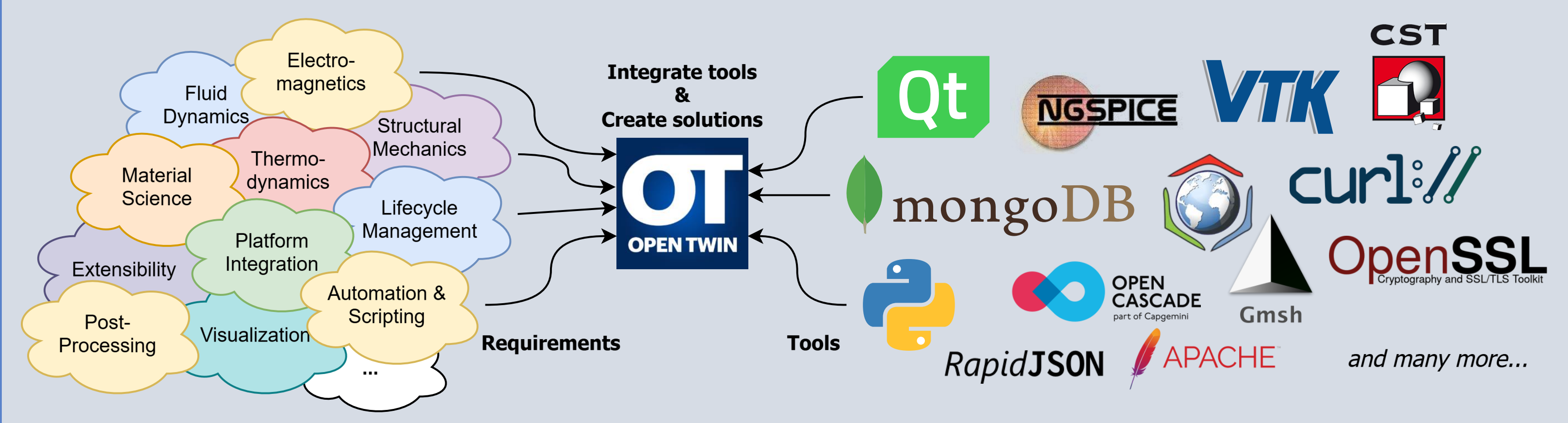
OpenTwin is an open-source platform that supports the development and operation of Digital Twins for product development. Its architecture is based on microservices, which are independently deployable, loosely coupled components (services) that communicate through lightweight protocols. This design makes OpenTwin flexible, scalable, and well-suited for integrating heterogeneous tools into digital twin workflows. The motivation for OpenTwin arises from pressing industrial challenges in multi domain and multiscale applications including structural mechanics, electromagnetics, thermodynamics, fluid dynamics, and material science, where accurate and scalable digital twin solutions are increasingly required. At present, OpenTwin supports the execution of individual, high-performance physical simulations.

Why OpenTwin?

Existing commercial digital twin solutions are often closed systems, which makes the integration of custom components and extensions difficult. In contrast, many academic or in-house solutions lack the robustness and infrastructure required for industrial applications. While a wide range of open-source tools exists for individual tasks, they typically do not address two central challenges:

1. **Platform integration** – OpenTwin provides a microservice-based framework for integrating heterogeneous tools into a distributed platform.
2. **Lifecycle management** – OpenTwin implements built-in dataset versioning and data management, enabling consistent tracking, undo/redo functionality, and branching.

By combining established open-source technologies within a unified architecture, OpenTwin bridges the gap between flexibility, extensibility, and industrial applicability.



Architecture Overview

The OpenTwin architecture is based on Rust-based web servers that dynamically load generic libraries (DLLs) to run microservices. This approach allows services to be implemented in different programming languages while remaining interoperable within a unified framework.
Communication between services is handled through HTTPS using JSON-based REST messages, supporting both synchronous and asynchronous interactions. At the core of the platform are seven main services:

1. **Authentication Service:** Manages user access to the database.
2. **Logging Service:** Collects and buffers log messages from all microservices.
3. **Global Session Service:** Coordinates Local Session Services and distributes session requests to the least loaded instance.
4. **Local Session Services:** Manages active sessions and balances load across nodes.
5. **Global Directory Service:** Oversees Local Directory Services and distributes microservice startup requests.
6. **Local Directory Service:** Launches and monitors microservices.
7. **Library Manager Service:** Manages data libraries stored in the database.

Beyond these core components, OpenTwin provides a broad range of specialized microservices for tasks such as **3D modeling, meshing, data processing, Python script execution, circuit simulation, and numerical solvers**. The interplay between the core services, user workstations, and domain-specific microservices is illustrated in the diagram below.

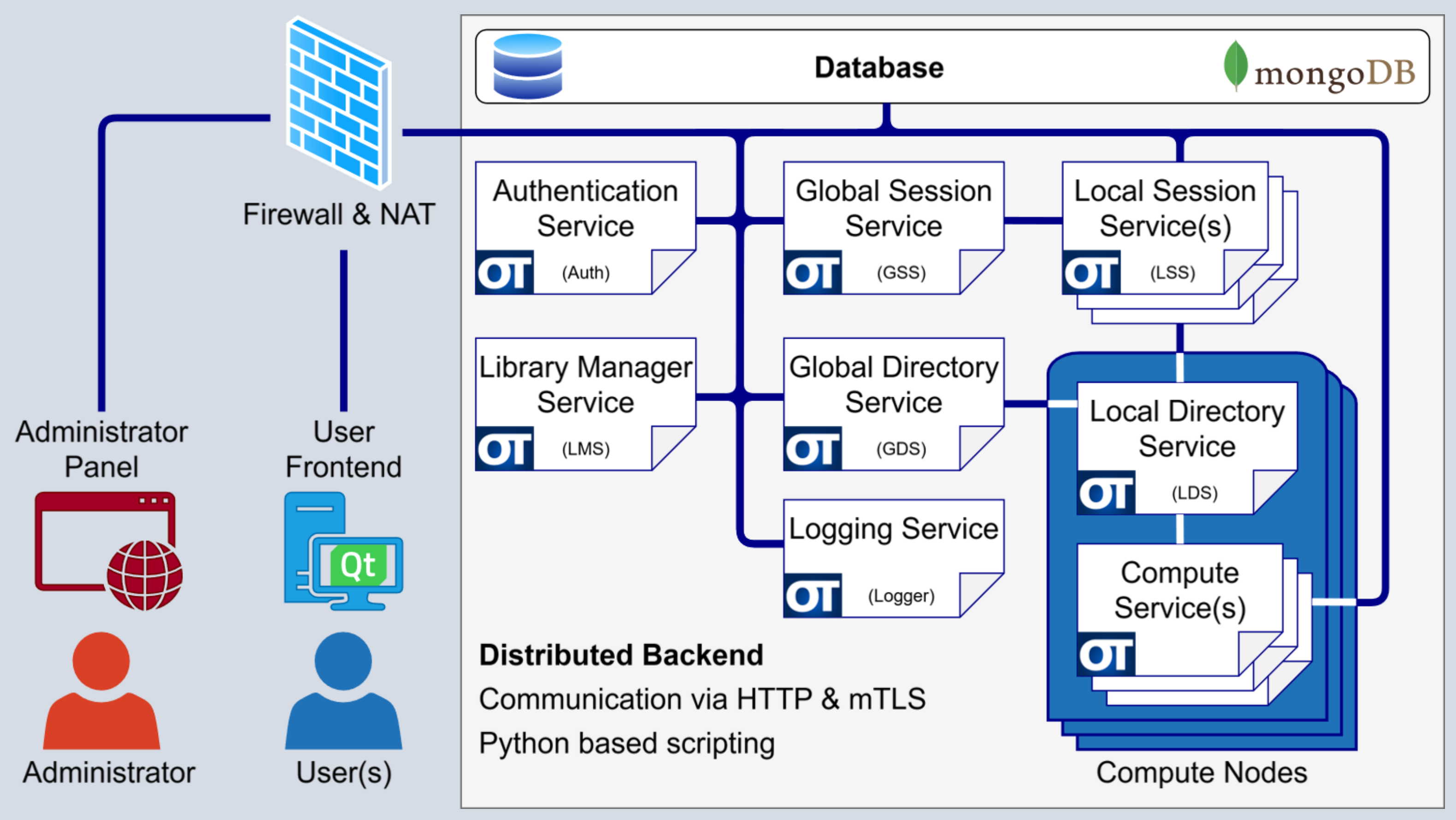
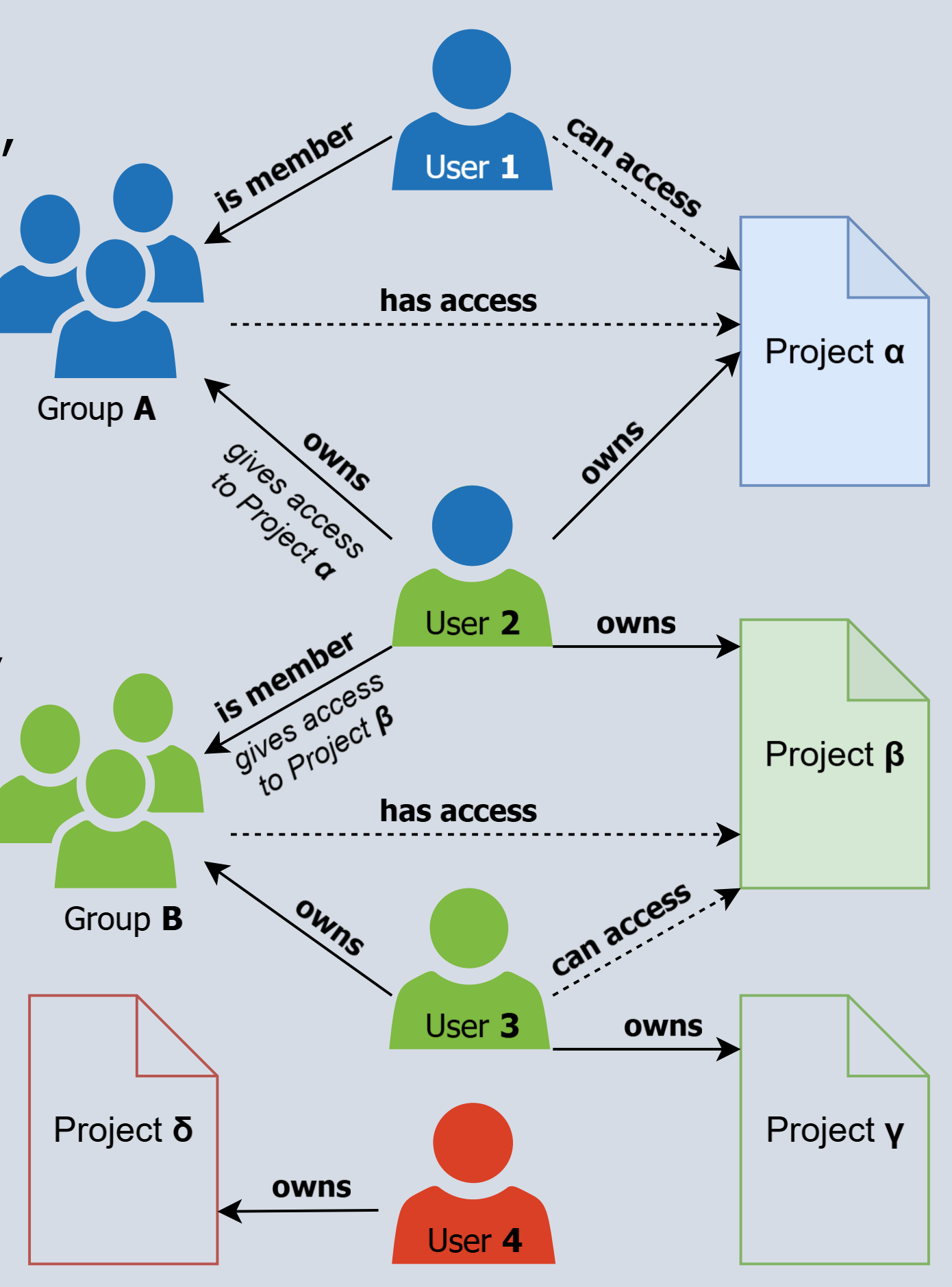

Database
mongoDB
Firewall & NAT
Authentication Service (Auth)
Global Session Service (GSS)
Local Session Service(s) (LSS)
Library Manager Service (LMS)
Global Directory Service (GDS)
Local Directory Service (LDS)
Logging Service (Logger)
Compute Service(s)
Administrator Panel
User Frontend
Administrator
User(s)
Distributed Backend
Communication via HTTP & mTLS
Python based scripting
Compute Nodes

Figure 1: OpenTwin Architecture Overview

OpenTwin supports multiple deployment scenarios: local workstations can connect to the backend, while compute-intensive tasks can be offloaded to clusters or cloud resources. This flexibility allows developers to focus on individual microservices without concerning themselves with the underlying infrastructure.
All data is managed in a MongoDB database as immutable, versioned objects, ensuring consistency, reproducibility, and full lifecycle tracking. In addition, OpenTwin exposes specialized APIs for tasks such as model management and visualization, which facilitate integration of domain-specific tools.

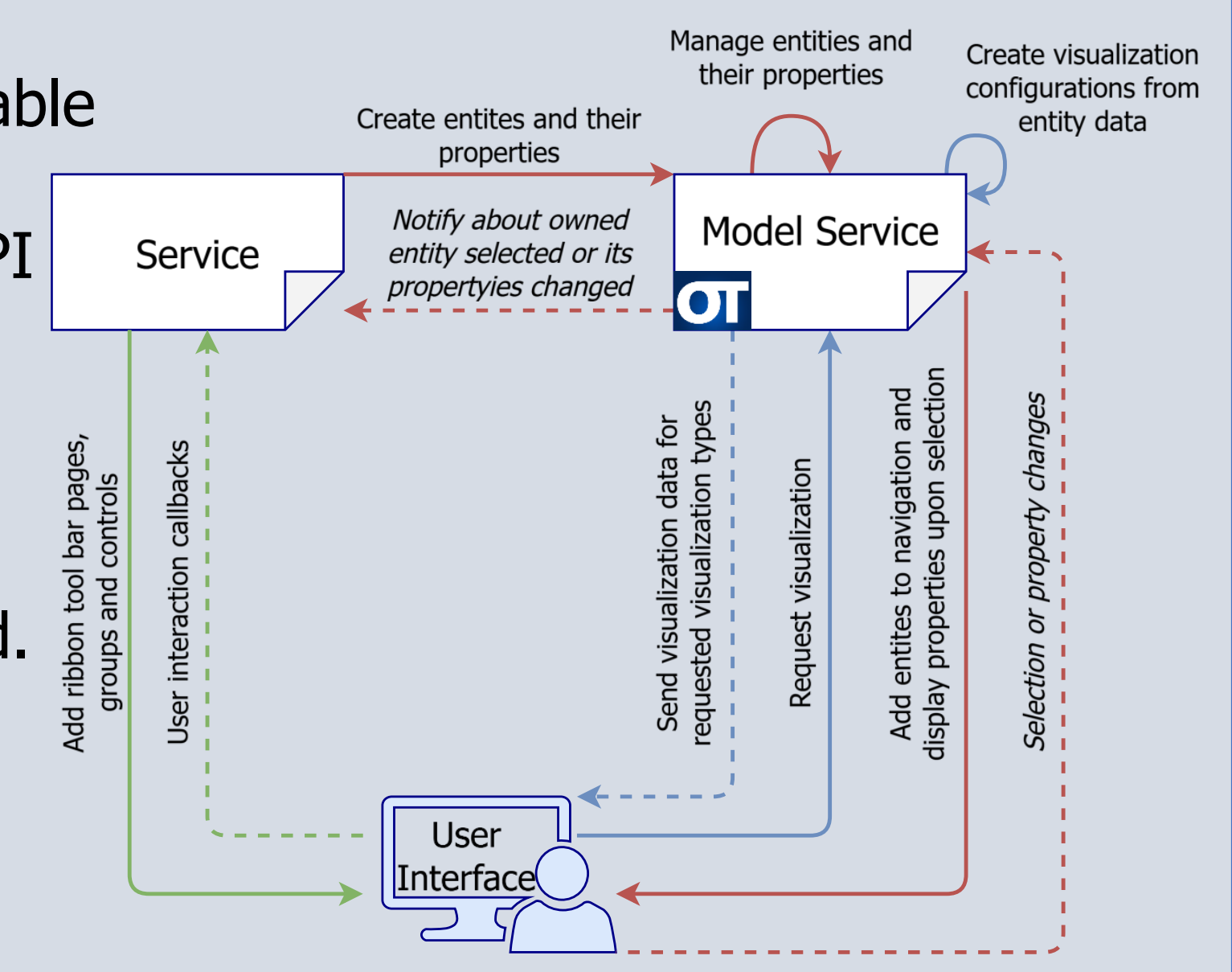
Project, User and Group Management

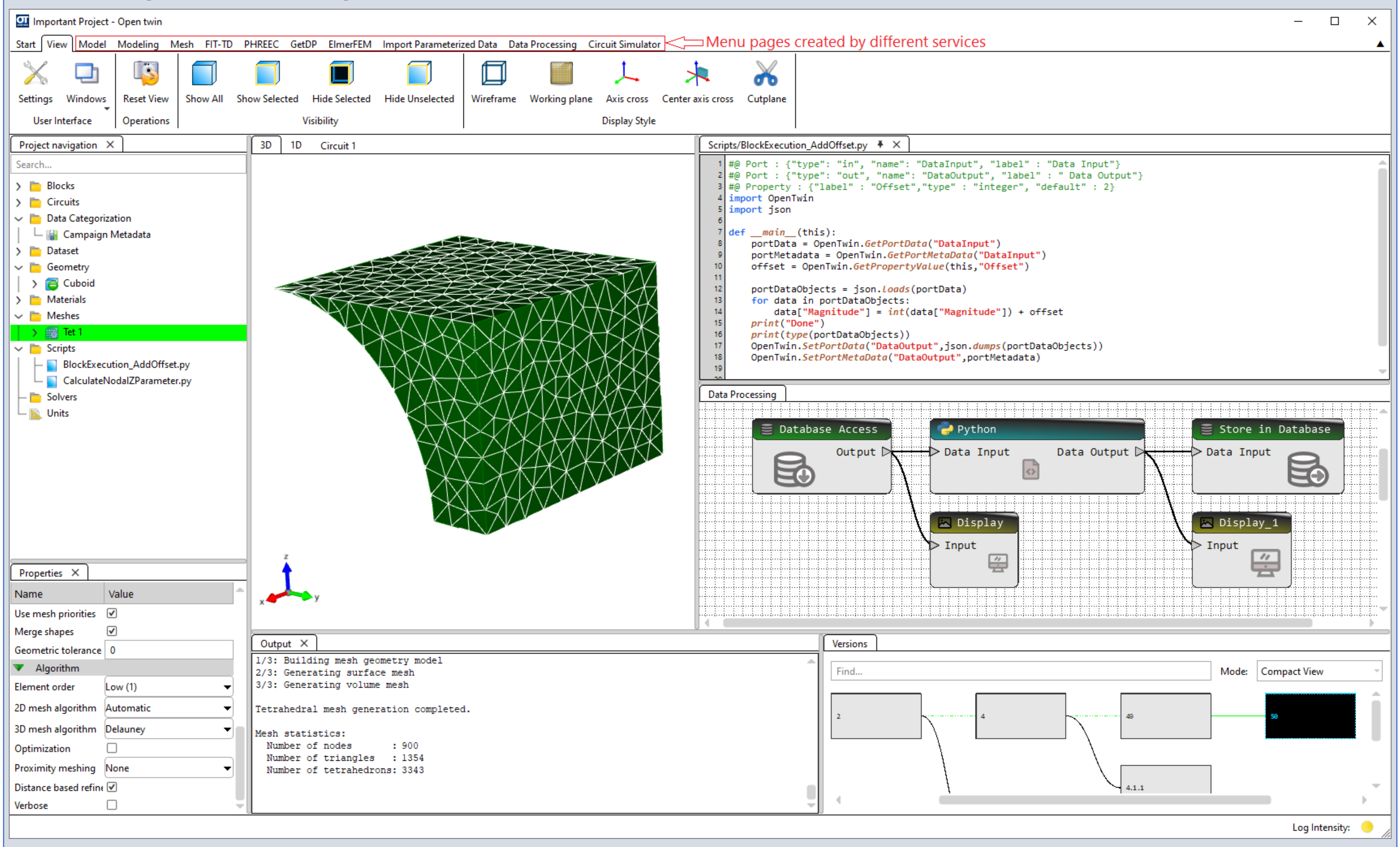
OpenTwin provides a structured framework for managing projects, users, and collaborative groups. Projects can be created for different application domains, such as 3D simulation workflows, data processing pipelines, or CST Studio Suite integrations. Each user maintains an individual project list, ensuring that personal workspaces and data remain clearly organized.
For collaborative use cases, OpenTwin supports the creation of user groups. Users can invite other participants into groups and grant shared access to selected projects. Once access is provided, all members of the group can open and work with the corresponding projects. Access control is strictly enforced: only the project owner is permitted to delete a shared project, although the ownership of a project can be explicitly transferred to another user when required.
Additional functionality supports project portability and reuse. Projects, whether personal or shared, can be copied into a user's workspace, allowing independent modifications without affecting the original. Furthermore, projects can be exported to a file and later imported into another OpenTwin environment, enabling efficient exchange of project configurations across users, groups, or installations.



Services Driven Frontend

The OpenTwin frontend is designed as a service-driven, scriptable interface that allows services to dynamically add, remove, or modify user interface elements during runtime. A dedicated API provides access to these functions, ensuring that services can flexibly adapt the frontend to specific workflows. User interactions generate callbacks that are routed directly to the responsible service for processing, while the frontend automatically performs cleanup when a service is disconnected.
A central component of the interface is the ribbon toolbar. Each service can register custom pages within the toolbar, which may contain multiple groups of controls. Within these groups, services can add common interface elements such as buttons, checkboxes, and other basic controls. This modular structure ensures a clear and consistent user experience, while allowing services to tailor the interface to their specific functionality.
In addition, the frontend provides a navigation tree and a property grid managed by the Model service. Each entity in a session is represented in the navigation tree, with its properties shown in the property grid upon selection. When a property is modified, the responsible service is automatically notified, enabling immediate synchronization between frontend and backend logic.
Entity visualization is handled directly by the frontend, which supports multiple visualization types. The appropriate visualization for a given entity is triggered automatically, ensuring consistent and efficient rendering across heterogeneous services.


Create entites and their properties
Manage entities and their properties
Create visualization configurations from entity data
Service
Notify about owned entity selected or its propertyies changed
Model Service
Add ribbon tool bar pages, groups and controls
User interaction callbacks
Send visualization data for requested visualization types
Request visualization
Add entites to navigation and display properties upon selection
Selection or property changes
User Interface



```
#@ Port : {"type": "in", "name": "DataInput", "label" : "Data Input"}
#@ Port : {"type": "out", "name": "DataOutput", "label" : " Data Output"}
#@ Property : {"label" : "Offset","type" : "integer", "default" : 2}
import OpenTwin
import json

def __main__(this):
    portData = OpenTwin.GetPortData("DataInput")
    portMetadata = OpenTwin.GetPortMetaData("DataInput")
    offset = OpenTwin.GetPropertyValue(this,"Offset")

    portDataObjects = json.loads(portData)
    for data in portDataObjects:
        data["Magnitude"] = int(data["Magnitude"]) + offset
    print("Done")
    print(type(portDataObjects))
    OpenTwin.SetPortData("DataOutput",json.dumps(portDataObjects))
    OpenTwin.SetPortMetaData("DataOutput",portMetadata)
```

```
1/3: Building mesh geometry model
2/3: Generating surface mesh
3/3: Generating volume mesh

Tetrahedral mesh generation completed.

Mesh statistics:
  Number of nodes       : 900
  Number of triangles   : 1354
  Number of tetrahedrons: 3343
```

Summary

OpenTwin is designed with extensibility as a core principle, enabling seamless adaptation to diverse application requirements. The platform provides multiple APIs for importing and exporting data from external tools, such as Dassault Systèmes CST Studio Suite, thereby facilitating integration with established industrial and academic workflows.
The microservice architecture ensures that existing services can be replaced by custom implementations, and new domain-specific microservices can be added to the existing pool with minimal effort. Dedicated APIs simplify the development process of custom services, while a frontend API allows developers to create user interface controls and handle corresponding callbacks.
In addition, Python scripting provides a flexible mechanism to orchestrate microservices through scripts, enabling automation, workflow customization, and rapid prototyping. Together, these features make OpenTwin a versatile platform that supports both standardized use cases and highly specialized extensions.

References

1. OpenTwin **GitHub** Repository: https://github.com/OT-OpenTwin
2. OpenTwin **Webpage**: opentwin.net

FRANKFURT UNIVERSITY OF APPLIED SCIENCES