



DTaaS

Digital Twin as a Service (DTaaS)

DTaaS Development Team

Copyright © 2022 - 2026 The INTO-CPS Association

Table of contents

1. What is the DTaaS Platform?	4
1.1 License	4
2. Media Playlist	5
2.1 Current GIF previews	5
2.2 Referenced slides and videos	6
3. User	9
3.1 DTaaS for Users	9
3.2 Overview	10
3.3 Website	12
3.4 Reusable Assets	39
3.5 Digital Twins	48
3.6 Working with GitLab	79
3.7 Runner	83
3.8 Examples	87
4. Admin	0
4.1 Installation Scenarios	0
4.2 DTaaS	0
4.3 Workspace	0
4.4 Platform Services	0
4.5 GitLab	0
4.6 Independent Packages	0
4.7 Guides	0
5. Frequently Asked Questions	0
5.1 Abbreviations	0
5.2 General Questions	0
5.3 Digital Twin Assets	0
5.4 Digital Twin Models	0
5.5 Communication Between Physical Twin and Digital Twin	0
5.6 Digital Twin DevOps Automation	0
5.7 Data Management	0
5.8 Platform Native Services on the DTaaS Platform	0
5.9 Comparison with other DT Platforms	0
5.10 GDPR Concerns	0
6. Developer	0
6.1 Guide	0

6.2 System	0
6.3 Technical Concepts	0
6.4 Codebase	0
6.5 Contributions	0
7. Known Issues in the Software	0
7.1 Third-Party Software	0
7.2 GitLab	0
8. Thanks	0
8.1 Funding Sources	0
8.2 Developers	0
8.3 Example Contributors	0
8.4 Documentation	0
9. License	0
9.1 License	0
9.2 Third Party Software	0

1. What is the DTaaS Platform?

The Digital Twin as a Service (DTaaS) software platform is designed to **Build, Use and Share** digital twins (DTs).

 **Build:** DTs are constructed on DTaaS using reusable DT assets available on the platform.

 **Use:** DTs can be executed on the DTaaS platform.

 **Share:** Ready-to-use DTs can be shared with other users. It is also possible to share the services offered by one DT with other users.

Here is a feature overview of the DTaaS v1.0 interface and workflow,

DTaaS v1.0 user walkthrough

The full-resolution recording is also available as [MP4](#).

An overview of the DTaaS platform available in the form of [slides](#) and a short [introduction video](#).

1.1 License

This software is owned by [The INTO-CPS Association](#) and is available under [the INTO-CPS License](#).

The DTaaS platform uses [third-party](#) open-source software. These software components have their own licenses.

2. Media Playlist

This page lists the slide decks and videos linked directly from the DTaaS documentation. GIF previews are embedded where preview files are available. When both standard and 4x assets exist, the 4x variant is shown here.

2.1 Current GIF previews

2.1.1 DTaaS v1.0 feature walkthrough

DTaaS v1.0 user walkthrough

Also see [MP4](#) version. Referenced from [index.md](#) and [user/motivation.md](#).

2.1.2 DTaaS on localhost: installation and walkthrough

DTaaS localhost walkthrough

Also see [MP4](#) version. Referenced from [admin/overview.md](#).

2.1.3 Multi-user DTaaS on secure server: installation and walkthrough

Multi-user DTaaS on secure server walkthrough

Also see [MP4](#) version. Referenced from [admin/overview.md](#).

2.1.4 Workspace on localhost: installation and walkthrough

Workspace on localhost walkthrough

Also see [MP4](#) version. Referenced from [admin/overview.md](#).

2.1.5 Multi-user workspace

Multi-user workspace walkthrough

Also see [MP4](#). Referenced from [admin/overview.md](#).

2.2 Referenced slides and videos

Topic	Slides/PDF	Video	Referenced From
DTaaS short introduction	Slides	Video	index.md
DTaaS v1.0 user walkthrough	-----	Video	index.md , user/motivation.md
User presentation	Slides	Video	user/motivation.md
Developer presentation	Slides	Video	developer/index.md
DTaaS DevOps workflows	-----	Video	developer/index.md
DTaaS and DevOps	-----	Video	developer/system/research-overview.md
Federated DTaaS	-----	Video	developer/system/research-overview.md
Automation layer demo	-----	Video	user/digital-twins/devops/ui.md
DTaaS localhost walkthrough	-----	Video	admin/overview.md
Multi-user DTaaS on secure server	-----	Video	admin/overview.md
Workspace on localhost walkthrough	-----	Video	admin/overview.md
Multi-user workspace walkthrough	-----	Video	admin/overview.md
Examples overview	Slides	Video	user/examples/index.md
CP-SENS project demo	Slides	Video	user/examples/index.md
Wind turbine blade testing demo	-----	Video	user/examples/index.md
Python package demo (CP-SENS)	-----	Video	user/examples/index.md
DAQ part 1	-----	Video	

Topic	Slides/PDF	Video	Referenced From
			user/examples/index.md
DAQ part 2	-----	Video	user/examples/index.md
Operational model analysis	-----	Video	user/examples/index.md
Model update demo	-----	Video	user/examples/index.md
Incubator demo	-----	Video	user/examples/index.md
MATLAB Simulink screencast	-----	Video	FAQ.md

3. User

3.1 DTaaS for Users

3.1.1 User Guide

This guide is intended for users of the DTaaS platform. Access to a live installation of the DTaaS platform is required. The simplest option is the [workspace localhost](#) installation scenario.

The following user-specific [Slides](#) and [conceptual presentation](#) provide the conceptual framework behind composable digital twins in the DTaaS platform.

3.1.2 Motivation

A central question in Digital Twin (DT) software platforms is how to enable collaborative activities:

- Building digital twins (DTs)
- Utilizing DTs independently
- Sharing DTs with other users
- Providing existing DTs as a service to other users

Additionally, DT software platforms must address:

- Support for DT lifecycle management
- Scalability through flexible convention over configuration

3.1.3 Existing Approaches

Several solutions have been proposed in recent literature to address these challenges. Notable approaches include:

- Focus on data from Physical Twins (PTs) for analysis, diagnosis, and planning
- Sharing DT assets across upstream and downstream stakeholders
- Evaluating different models of PT
- DevOps methodologies for Cyber Physical Systems (CPS)
- Scaling DT execution and ensemble management of related DTs
- Support for PT product lifecycle

3.1.4 The DTaaS Approach

The DTaaS platform adopts the following principles[1]:

- Support for transition from existing workflows to DT frameworks
- Creation of DTs from reusable assets
- Enabling users to share DT assets
- Offering DTs as a Service
- Integration of DTs with external software systems
- Separation of configurations for independent DT components

3.1.5 References

[1]: Talasila, Prasad, et al. "Composable digital twins on Digital Twin as a Service platform." *Simulation* 101.3 (2025): 287-311.

3.2 Overview

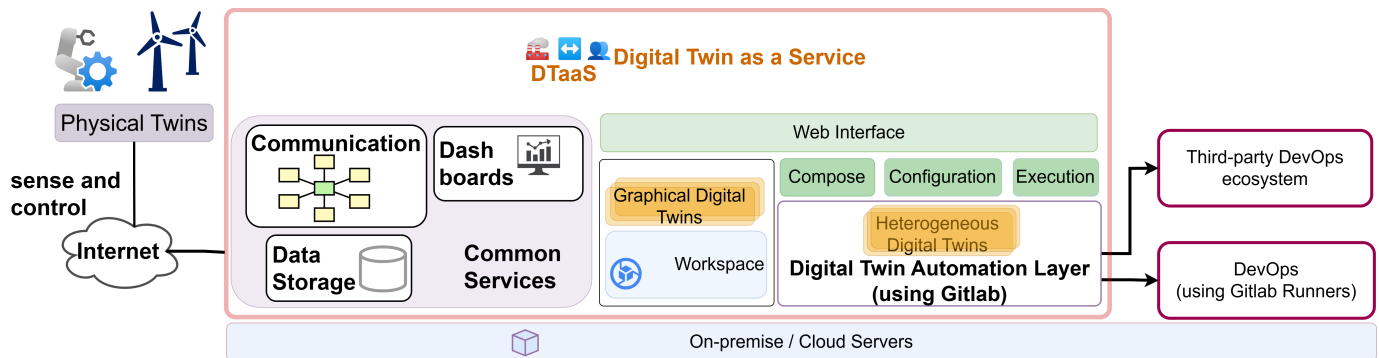
3.2.1 Advantages

The DTaaS software platform provides the following advantages:

- Support for heterogeneous Digital Twin implementations
- CFD, Simulink, co-simulation, FEM, ROM, ML, and other paradigms
- Integration with existing Digital Twin frameworks
- Provision of Digital Twin as a Service capabilities
- Facilitation of collaboration and asset reuse
- Private workspaces for verification of reusable assets and trial executions of DTs
- Cost effectiveness through shared infrastructure

3.2.2 Software Features

Each installation of the DTaaS platform includes the features illustrated in the following diagram.



All users are provided with dedicated workspaces. These workspaces are containerized implementations of Linux Desktops. The user desktops are isolated, ensuring that installations and customisations performed in one workspace do not affect other user workspaces. Graphical digital twins can be executed within these private workspaces.

Each user workspace is provisioned with pre-installed development tools. These tools are accessible directly through a web browser. The following tools are currently available:

Tool	Advantage
Jupyter Lab	Enables flexible creation and use of digital twins and their components through a web browser. All native JupyterLab use cases are supported.
Jupyter Notebook	Facilitates web-based management of files and library assets.
VS Code in the browser	A widely-adopted IDE for software development. Digital twin-related assets can be developed within this environment.
ungit	An interactive git client enabling repository management through a web browser.

In addition, an xfce-based remote desktop is accessible via a VNC client. The VNC client is available directly in the web browser. Desktop software supported by xfce can also be executed within the workspace.

The workspaces maintain Internet connectivity, enabling Digital Twins running in the workspace to interact with both internal and external services.

A DT automation layer is provided for managing DT automation tasks. This layer facilitates creation, modification, and execution of DTs on both on-premise infrastructure and commercial cloud (DevOps) service providers.

The DTaaS software platform includes several pre-installed services. The currently available services are:

Service	Advantage
InfluxDB	Internet of Things (IoT) device management and data visualisation platform. This service stores data for digital twins and provides alerting capabilities.
RabbitMQ	Communication broker facilitating message exchange between physical and digital twins.
Grafana	visualisation dashboard service for digital twin data presentation.
MQTT	Lightweight data transfer broker for IoT devices and physical twins providing data to digital twins.
MongoDB	NoSQL document database for storing metadata from physical twins.
PostgreSQL	SQL database server for storing historical and time-series data.
ThingsBoard	an Internet of Things (IoT) device management and data visualisation platform

Users can publish and reuse digital twin assets available on the platform. Additionally, digital twins can be executed and made available as services to external clients. These clients need not be registered users of the DTaaS installation.

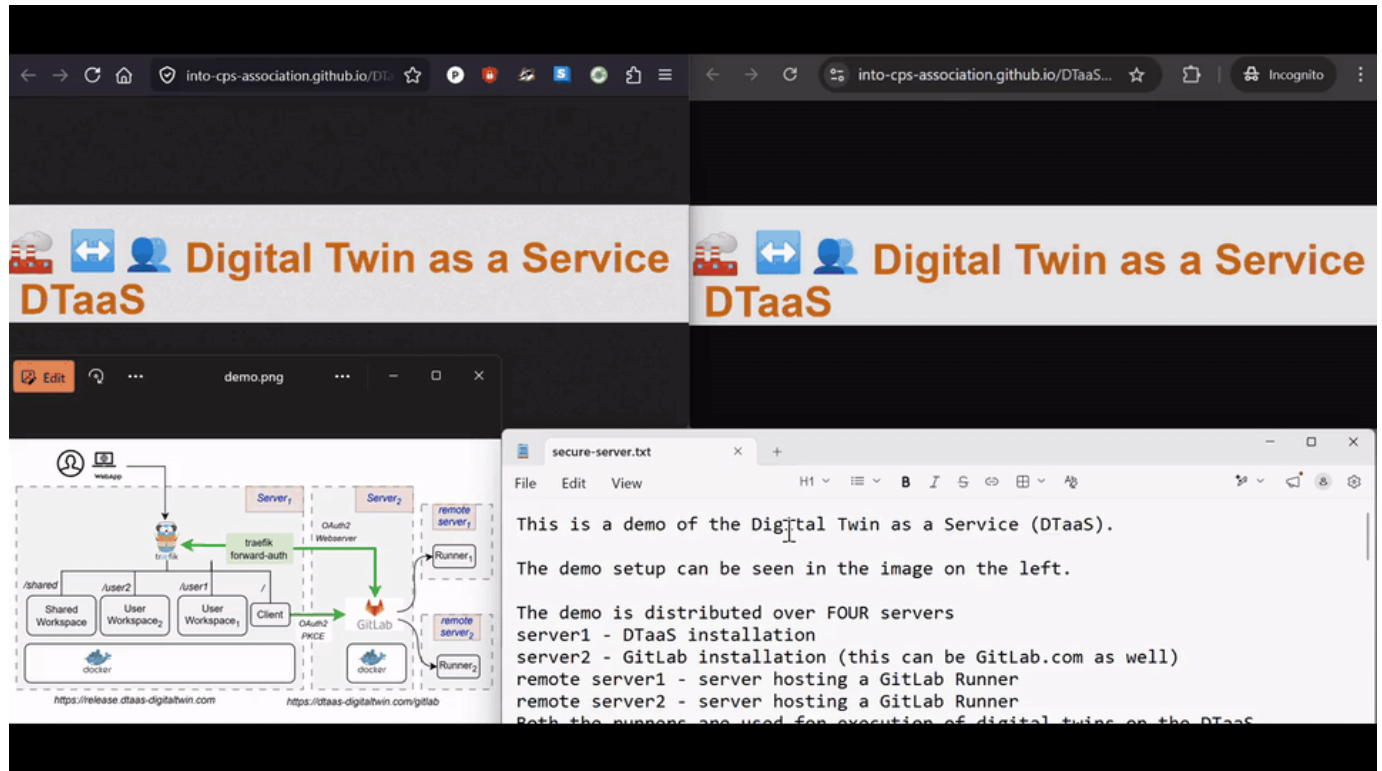
3.2.3 References

Talasila, Prasad, et al. "Composable digital twins on Digital Twin as a Service platform." Simulation 101.3 (2025): 287-311.

3.3 Website

3.3.1 DTaaS Website Screenshots

This page provides a screenshot-driven preview of the website serving the DTaaS software platform. A quick screen recording is available as well.

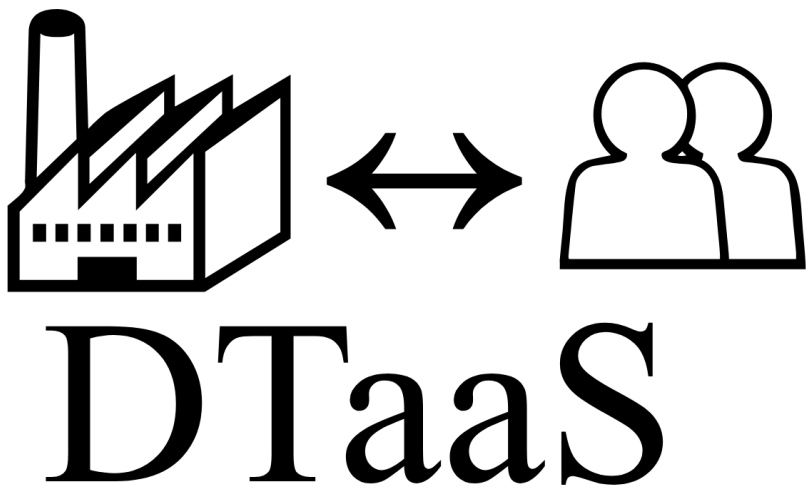


The <https://intocps.org> is used in this tutorial as an illustration for base URL of the installation. Do replace it with the correct URL of the installation.

Visit the DTaaS Installation

URL: <https://intocps.org> (specified in `config/client.js` file)

Navigation begins by visiting the website of the DTaaS instance for which the user is registered.



Redirected to Authorization Provider

URL: `https://gitlab.com/xxxx` (specified in `config/.env` file)

The browser redirects to the GitLab Authorization page for the DTaaS.



GitLab Community Edition

Username or primary email

TestUserDTaaS

Password

●●●●●●●●●●●●●●●●

[Forgot your password?](#)

☐ Remember me

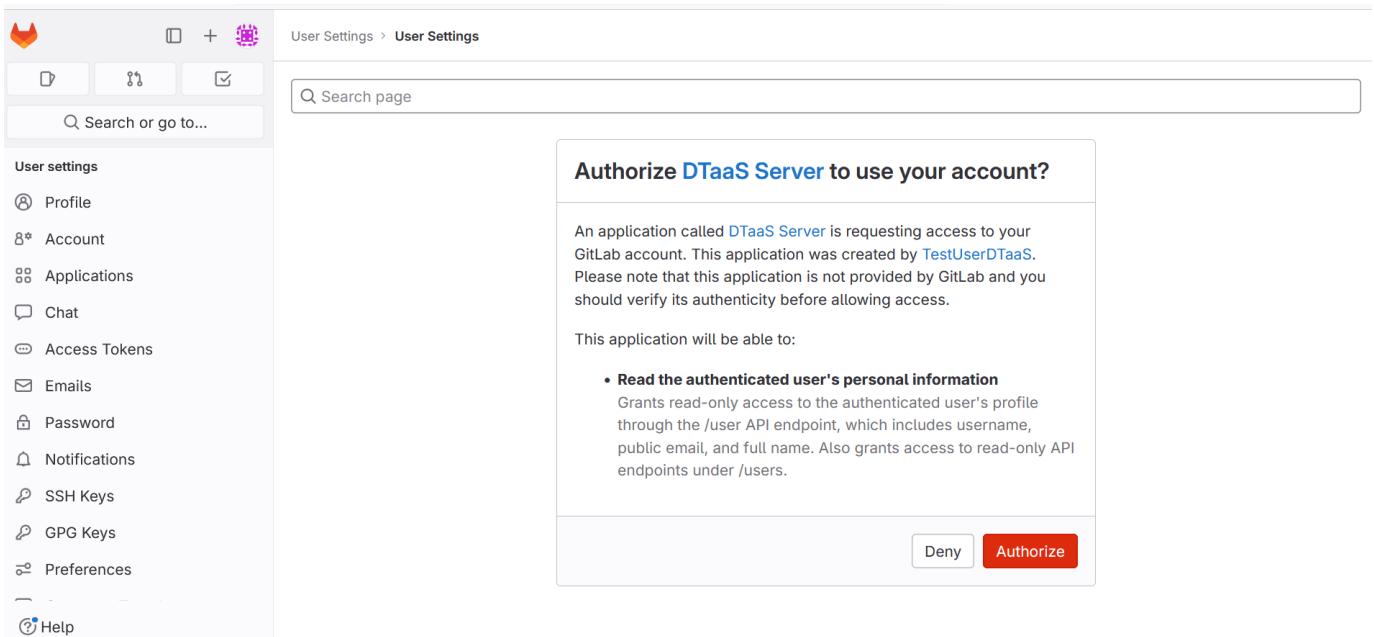
Sign in

The email/username and password should be entered. If the email ID registered with the DTaaS matches a GitLab Login email ID.

The browser redirects to the OAuth 2.0 Application page.

Permit DTaaS Server to Use GitLab

URL: `https://gitlab.com/xxxx`

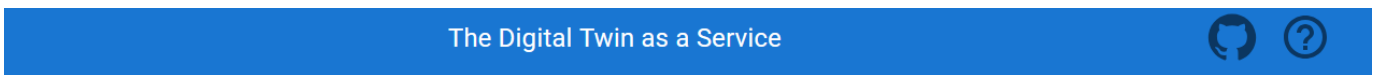


Clicking on Authorize permits the OAuth 2.0 application to access the information associated with the GitLab account. This is a required step.

After successful authentication, redirection to the login page of the DTaaS website occurs.

URL: <https://intocps.org>

The DTaaS website employs an additional layer of security - the third-party authorization protocol known as [OAuth 2.0](#). This protocol provides secure access to a DTaaS installation for users with active accounts at the selected OAuth 2.0 service provider. This implementation also uses GitLab as the OAuth 2.0 provider.

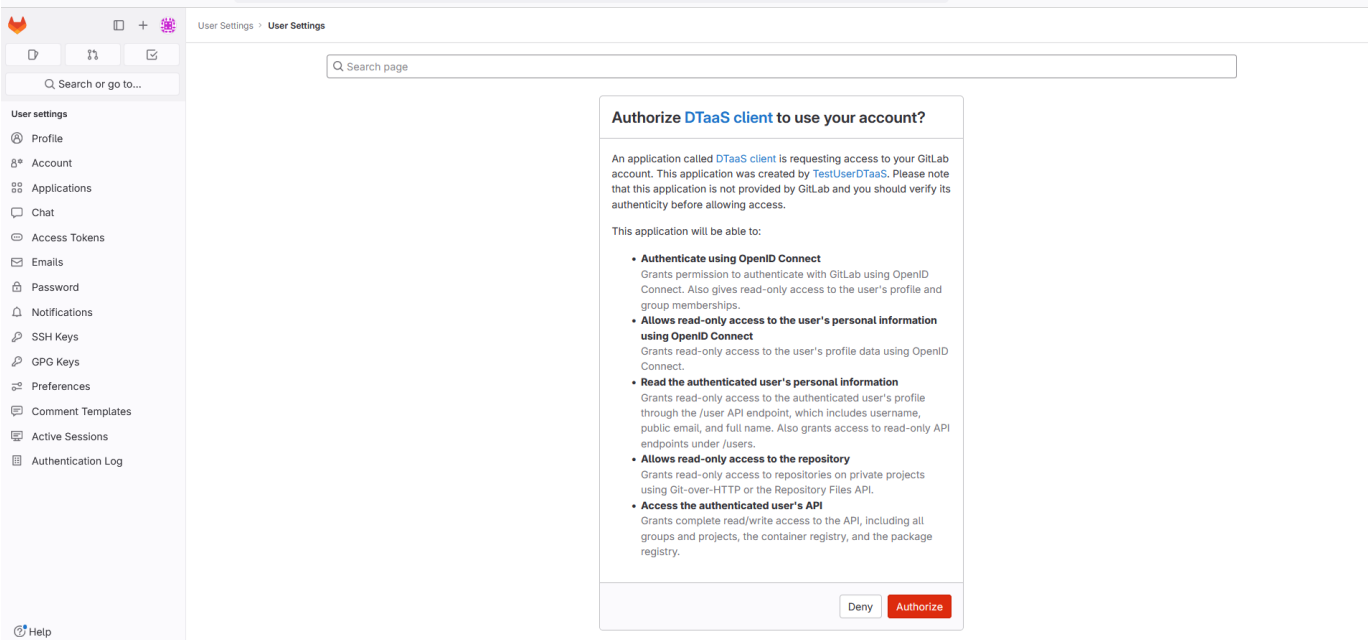


The GitLab signin button is displayed. Clicking this button redirects to the GitLab instance providing authorization for DTaaS. Re-authentication to GitLab is not required, unless explicit logout from the GitLab account has occurred.

Permit DTaaS Website to Use GitLab

URL: `https://gitlab.com/xxxx`

The DTaaS website requires permission to use the GitLab account for authorization. The **Authorize** button must be clicked.



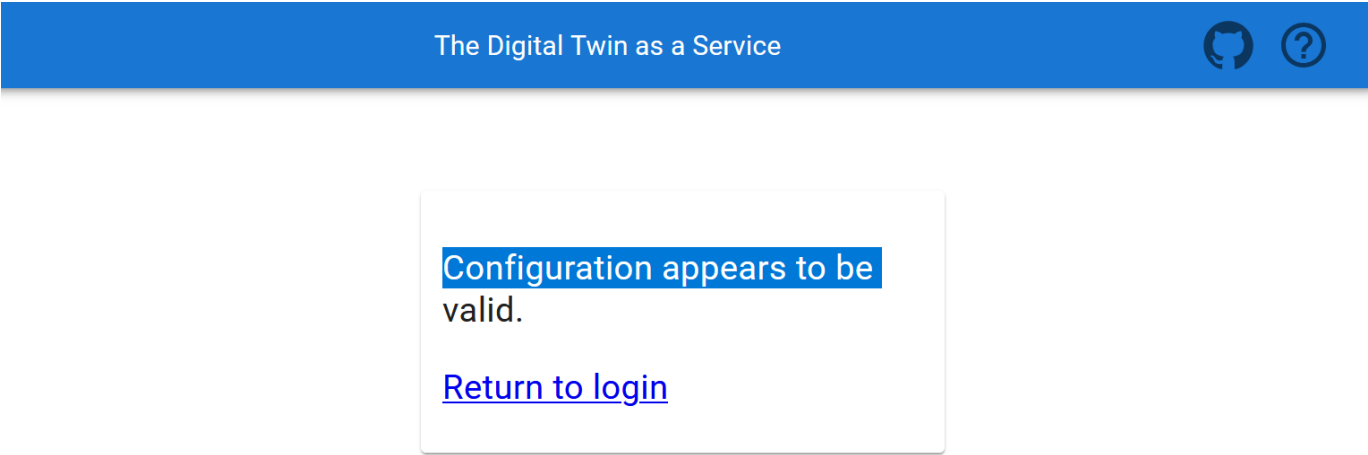
After successful authorization, redirection to the **Library** page of the DTaaS website occurs.

URL: `https://intocps.org/library/xxxx`

Three icons are located on the top-right of the webpage. The hyperlink on the **?** icon redirects to the help page, while the hyperlink on the **github icon** redirects to the GitHub code repository. The **Account** using the top-right purple **A** icon provides options to [change settings](#) and logout.

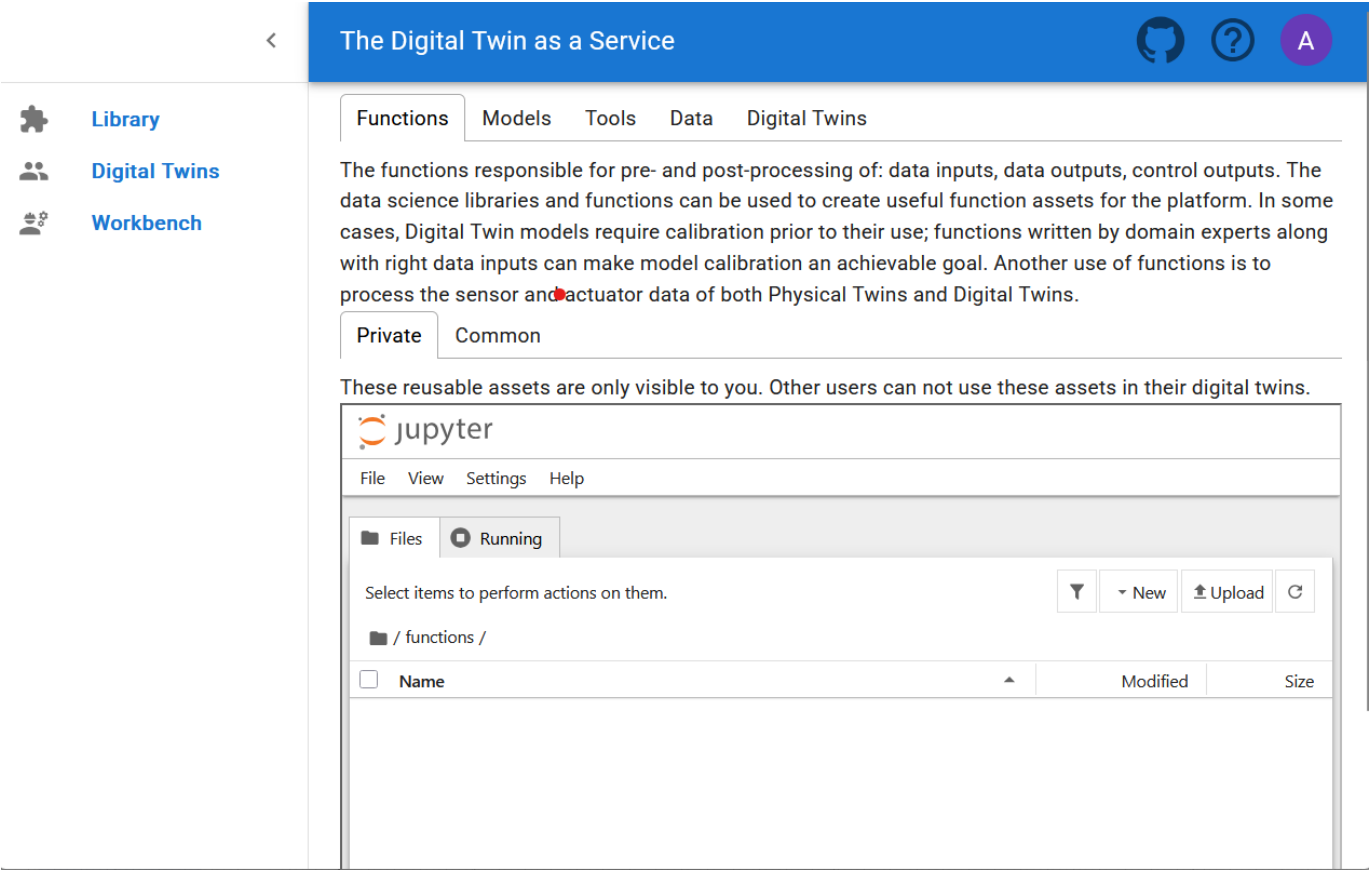
Check Website Access

For troubleshooting login issues, the website configuration can be verified by navigating to <https://intocps.org/config/user>. The following display indicates a correctly configured application.



Menu Items

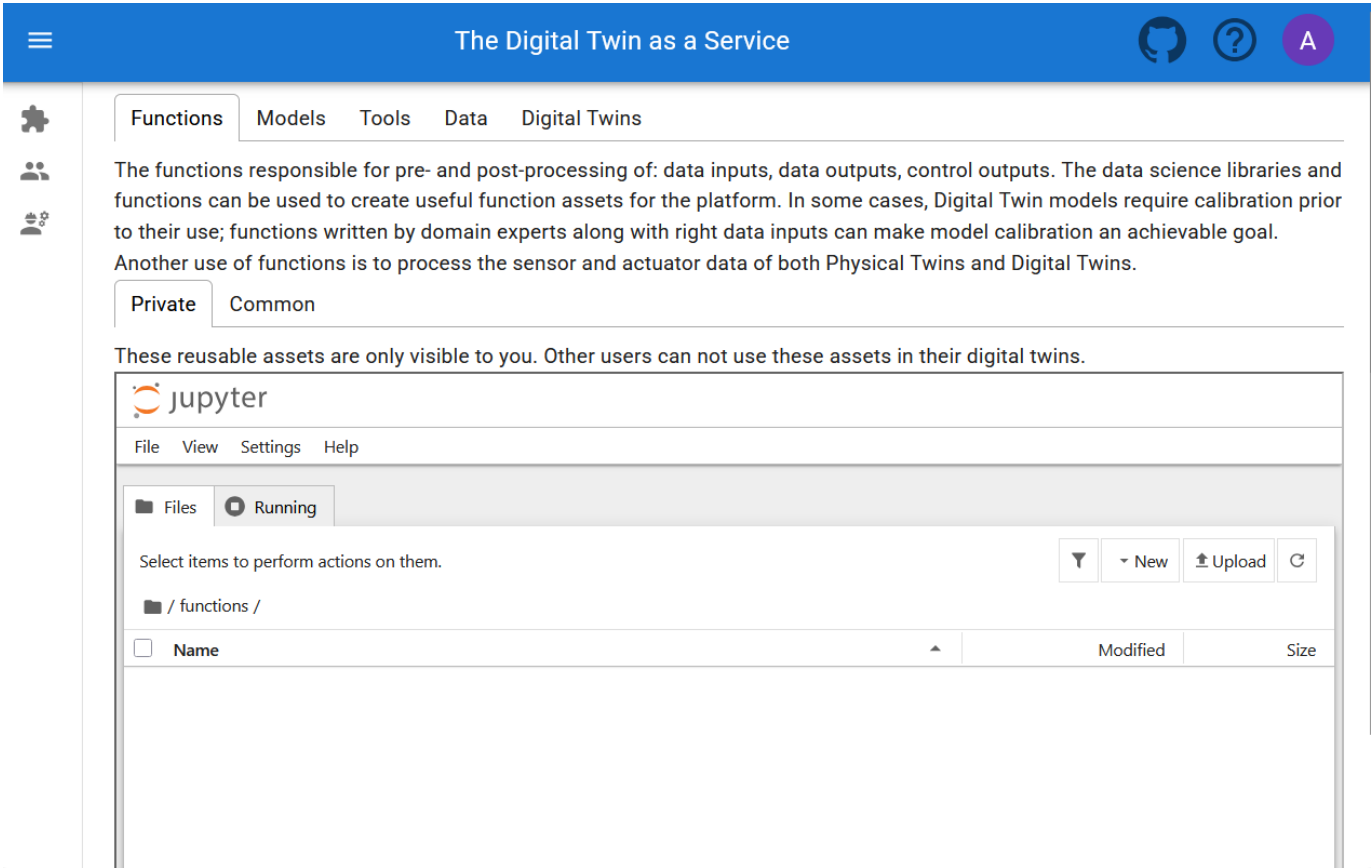
The menu is hidden by default. Only the icons of menu items are visible. Clicking on the **three horizontal bars** icon in the top-left corner of the page reveals the menu.



Three menu items are available:

- Library:** For management of reusable library assets. Files can be uploaded, downloaded, created, and modified on this page.
- Digital Twins:** For management of digital twins. A Jupyter Lab page is presented from which digital twins can be executed.
- Workbench:** Not all digital twins can be managed within Jupyter Lab. Additional tools are available on this page.

Library Page



Five tabs are displayed, each corresponding to one type of digital twin asset. Each tab provides help text to guide users on the asset type.

Functions

The functions responsible for pre- and post-processing of: data inputs, data outputs, control outputs. The data science libraries and functions can be used to create useful function assets for the platform. In some cases, Digital Twin models require calibration prior to their use; functions written by domain experts along with right data inputs can make model calibration an achievable goal. Another use of functions is to process the sensor and actuator data of both Physical Twins and Digital Twins.

Data

The data sources and sinks available to a digital twins. Typical examples of data sources are sensor measurements from Physical Twins, and test data provided by manufacturers for calibration of models. Typical examples of data sinks are visualisation software, external users and data storage services. There exist special outputs such as events, and commands which are akin to control outputs from a Digital Twin. These control outputs usually go to Physical Twins, but they can also go to another Digital Twin.

Models

The model assets are used to describe different aspects of Physical Twins and their environment, at different levels of abstraction. Therefore, it is possible to have multiple models for the same Physical Twin. For example, a flexible robot used in a car production plant may have structural model(s) which will be useful in tracking the wear and tear of parts. The same robot can have a behavioural model(s) describing the safety guarantees provided by the robot manufacturer. The same robot can also have a functional model(s) describing the part manufacturing capabilities of the robot.

Tools

The software tool assets are software used to create, evaluate and analyse models. These tools are executed on top of a computing platforms, i.e., an operating system, or virtual machines like Java virtual machine, or inside docker containers. The tools tend to be platform specific, making them less reusable than models. A tool can be packaged to run on a local or distributed virtual machine environments thus allowing selection of most suitable execution environment for a Digital Twin. Most models require tools to evaluate them in the context of data inputs. There exist cases where executable packages are run as binaries in a computing environment. Each of these packages are a pre-packaged combination of models and tools put together to create a ready to use Digital Twins.

Digital Twins

These are ready to use digital twins created by one or more users. These digital twins can be reconfigured later for specific use cases.

Two sub-tabs exist: **private** and **common**. Library assets in the private category are visible only to the logged-in user, while library assets in the common category are available to all users.

Further explanation on the placement of reusable assets within each type and the underlying directory structure on the server is available on the [assets page](#).

Note

Assets (files) can be uploaded using the **upload** button.

i The file manager is based on Jupyter Notebook, and all tasks available in Jupyter Notebook can be performed here.

Digital Twins Page

URL: <https://intocps.org/digitaltwins>

The screenshot shows the 'The Digital Twin as a Service' web application. The interface has a blue header with a menu icon, the title 'The Digital Twin as a Service', and user profile icons. On the left, there is a sidebar with icons for settings, users, and a file manager. The main content area has three tabs: 'Create', 'Execute', and 'Analyze'. Below the tabs, there is a text instruction: 'Create digital twins from tools provided within user workspaces. Each digital twin will have one directory. It is suggested that user provide one bash shell script to run their digital twin. Users can create the required scripts and other files from tools provided in Workbench page.' Below this instruction is a Jupyter Lab interface. The Jupyter Lab interface has a top bar with 'File', 'Edit', 'View', 'Run', 'Kernel', 'Tabs', 'Settings', and 'Help'. The left pane shows a file browser with a search bar and a table of files. The right pane shows a 'Launcher' with a 'Notebook' icon and a 'Console' icon. The bottom status bar shows 'Simple' mode, '0' files, and 'Mem: 96.03 / 7751.00 MB'. At the bottom of the page, there is a copyright notice: 'Copyright © The INTO-CPS Association 2024.' and a thank you note: 'Thanks to Material-UI for the Dashboard template, which was the basis for our React app.'

The digital twins page contains three tabs, and the central pane opens Jupyter Lab. The three tabs provide helpful instructions on suggested tasks for the **Create - Execute - Analyze** lifecycle phases of a digital twin. More explanation is available on the [lifecycle phases of digital twin](#).

Create

Create digital twins from tools provided within user workspaces. Each digital twin will have one directory. It is suggested that user provide one bash shell script to run their digital twin. Users can create the required scripts and other files from tools provided in Workbench page.

Execute

Digital twins are executed from within user workspaces. The given bash script gets executed from digital twin directory. Terminal-based digital twins can be executed from VSCode and graphical digital twins can be executed from VNC GUI. The results of execution can be placed in the data directory.

Analyze

The analysis of digital twins requires running of digital twin script from user workspace. The execution results placed within data directory are processed by analysis scripts and results are placed back in the data directory. These scripts can either be executed from VSCode and graphical results or can be executed from VNC GUI.

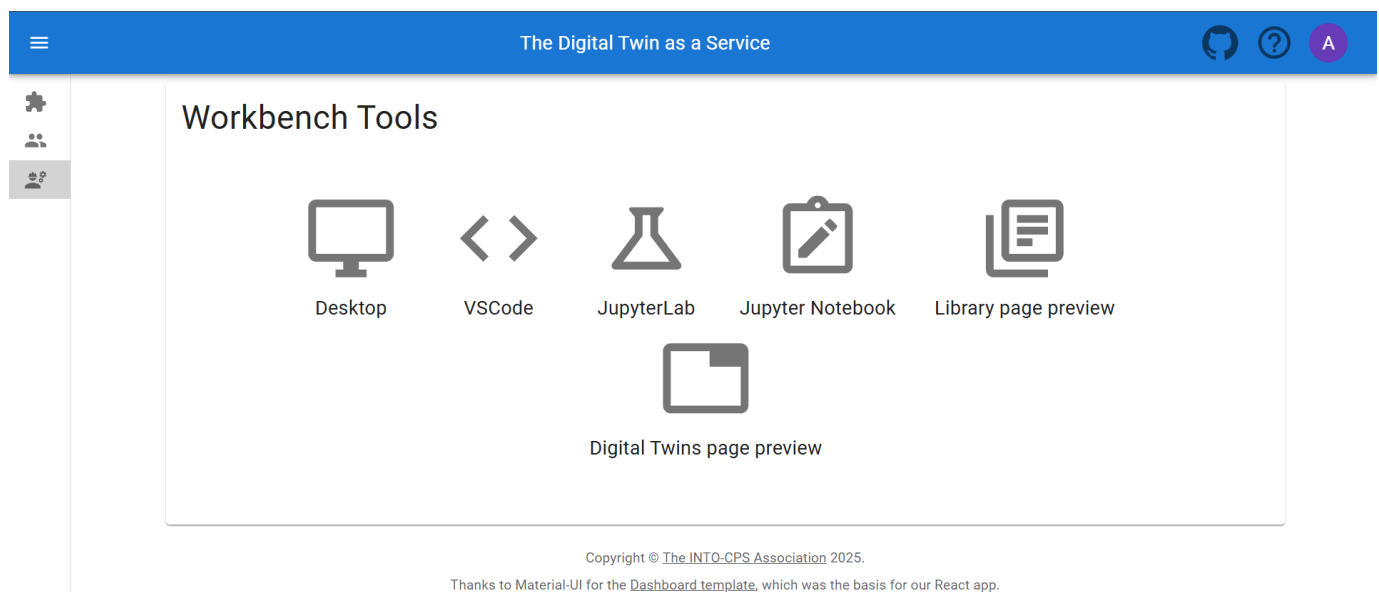
i The reusable assets (files) displayed in the file manager are also available in Jupyter Lab. Additionally, a git plugin is installed in Jupyter Lab that enables linking files with external git repositories.

Workbench

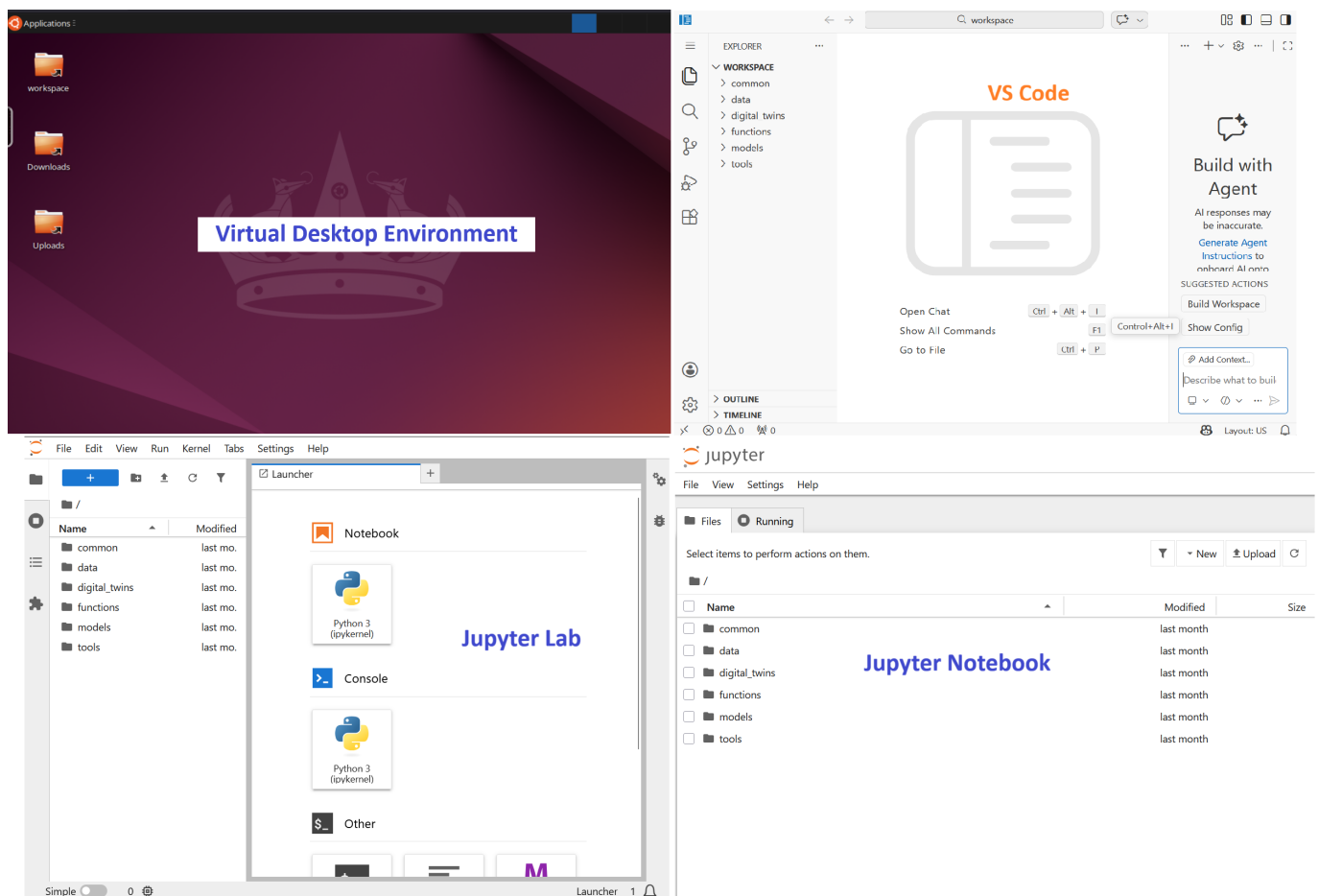
URL: <https://intocps.org/workbench>

The **workbench** page provides links to six integrated tools:

- Desktop
- VS Code
- Jupyter Lab
- Jupyter Notebook
- Library page preview
- Digital Twins page preview



Screenshots of the pages opened by clicking on first four icons are shown:



The hyperlinks open in new browser tabs.

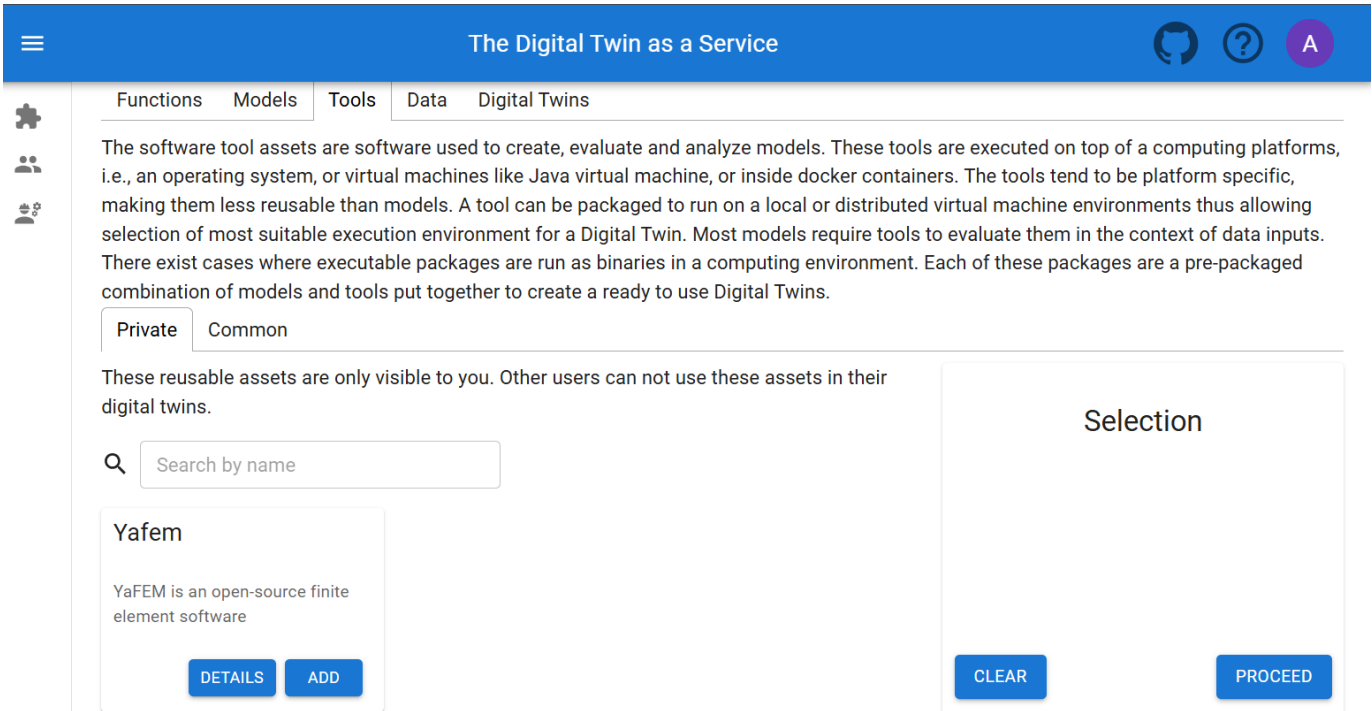
The **workbench** also has two links to DevOps-based implementation of composable digital twins.

- Library Page Preview
- Digital Twins Page Preview

LIBRARY PREVIEW PAGE

URL: <https://intocps.org/preview/library>

This page has the same philosophy of [Library page](#) and provides similar user interface.



Unlike the Library page, this preview page uses digital twin assets stored in a GitLab repository. New digital twins can be composed by selecting the required library assets.

The screenshot shows the 'The Digital Twin as a Service' web application. The interface includes a top navigation bar with a menu icon, the title 'The Digital Twin as a Service', and icons for GitHub, help, and a user profile. The left sidebar contains icons for a puzzle piece, users, and a gear. The main content area features tabs for 'Functions', 'Models', 'Tools', 'Data', and 'Digital Twins'. Below the tabs, a description states: 'These are ready to use digital twins created by one or more users. These digital twins can be reconfigured later for specific use cases.' There are two buttons: 'Private' (selected) and 'Common'. A text block indicates: 'These reusable assets are only visible to you. Other users can not use these assets in their digital twins.' Below this is a search bar labeled 'Search by name'. Five asset cards are displayed: 'Hello world', 'Mass spring damper', 'Runner', 'Shm devops', and 'Yafem demo'. Each card includes a description and buttons for 'DETAILS' and 'ADD' (except 'Mass spring damper' which has 'REMOVE'). On the right, a 'Selection' box with a red border contains a list item 'digital_twins/mass-spring-damper' and two buttons: 'CLEAR' and 'PROCEED'.

Upon clicking **Proceed** button, the digital twins create tab is opened.

Digital Twins Preview Page

URL: <https://intocps.org/preview/digitaltwins>

The **Digital Twins Preview Page** provides means of managing digital twins using the DevOps methodology. This page has three tabs, namely **Create**, **Manage** and **Execute**.

CREATE TAB

The library assets selected will be used on the **Create Tab** for creating new digital twins. The new digital twins are saved in the linked GitLab repository. Remember to add valid `.gitlab-ci.yml` configuration as it is used for execution of digital twin.

The Digital Twin as a Service

This page demonstrates integration of DTaaS with GitLab CI/CD workflows. The feature is experimental and requires certain GitLab setup in order for it to work.

Create

Manage

Execute

Create and save new digital twins. The new digital twins are saved in the linked gitlab repository. Remember to add valid '.gitlab-ci.yml' configuration as it is used for execution of digital twin.

Insert digital twin name

Composite-DT

ADD NEW FILE

DELETE FILE

RENAME FILE

▼ Description

description.md

README.md

▼ Configuration

.gitlab-ci.yml

config.json

▼ Lifecycle

execute

▼ mass-spring-damper

configuration

.gitlab-ci.yml

cosim.json

time.json

EDITOR

PREVIEW

1 image: ubuntu:20.04

2

3 stages:

4 - create

5 - execute

6 - clean

7

8 create_mass-spring-damper:

9 stage: create

10 script:

11 - cd digital_twins/mass-spring-damper

12 - chmod +x lifecycle/create

13 - lifecycle/create

14 tags:

15 - \$RunnerTag

16

17 execute_mass-spring-damper:

18 stage: execute

19 script:

20 - cd digital_twins/mass-spring-damper

21 - chmod +x lifecycle/execute

CANCEL

SAVE

MANAGE TAB

Complete descriptions of digital twins can be read.

The Digital Twin as a Service

This page demonstrates integration of DTaaS with GitLab CI/CD workflows. The feature is experimental and requires certain GitLab setup in order for it to work.

Create Manage Execute

Explore, edit and delete existing digital twins. The changes get saved in the linked gitlab repository.

Search by name

Hello world

The hello world digital twin (DT) is a simple demonstrative model designed to introduce the basic concepts of a DT.

DETAILS RECONFIGURE DELETE

Mass spring damper

The mass spring damper digital twin (DT) comprises two mass spring dampers and demonstrates how a co-simulation based DT can be used within

DETAILS RECONFIGURE DELETE

Runner

Use DT Runner to execute commands.

DETAILS RECONFIGURE DELETE

Shm devops

This project creates a fully-functional demo of CP-SENS project.

DETAILS RECONFIGURE DELETE

Yafem demo

Execute YaFEM demo in devops pipeline of DTaaS.

DETAILS RECONFIGURE DELETE

Copyright © The INTO-CPS Association 2026.
Thanks to Material-UI for the [Dashboard template](#), which was the basis for our React app.

If necessary, a digital twin can be deleted, removing it from the workspace along with all associated data. Digital twins can also be reconfigured.

The Digital Twin as a Service

This page demonstrates integration of DTaaS with GitLab CI/CD workflows. The feature is experimental and requires certain GitLab setup in order for it to work.

Create Manage Execute

Reconfigure Mass spring damper

- ▼ Description
 - README.md
 - description.md
- ▼ Configuration
 - .gitlab-ci.yml
 - cosim.json
 - time.json
- ▼ Lifecycle
 - analyze
 - clean
 - create
 - evolve
 - execute
 - save
 - terminate

Edit Files

```

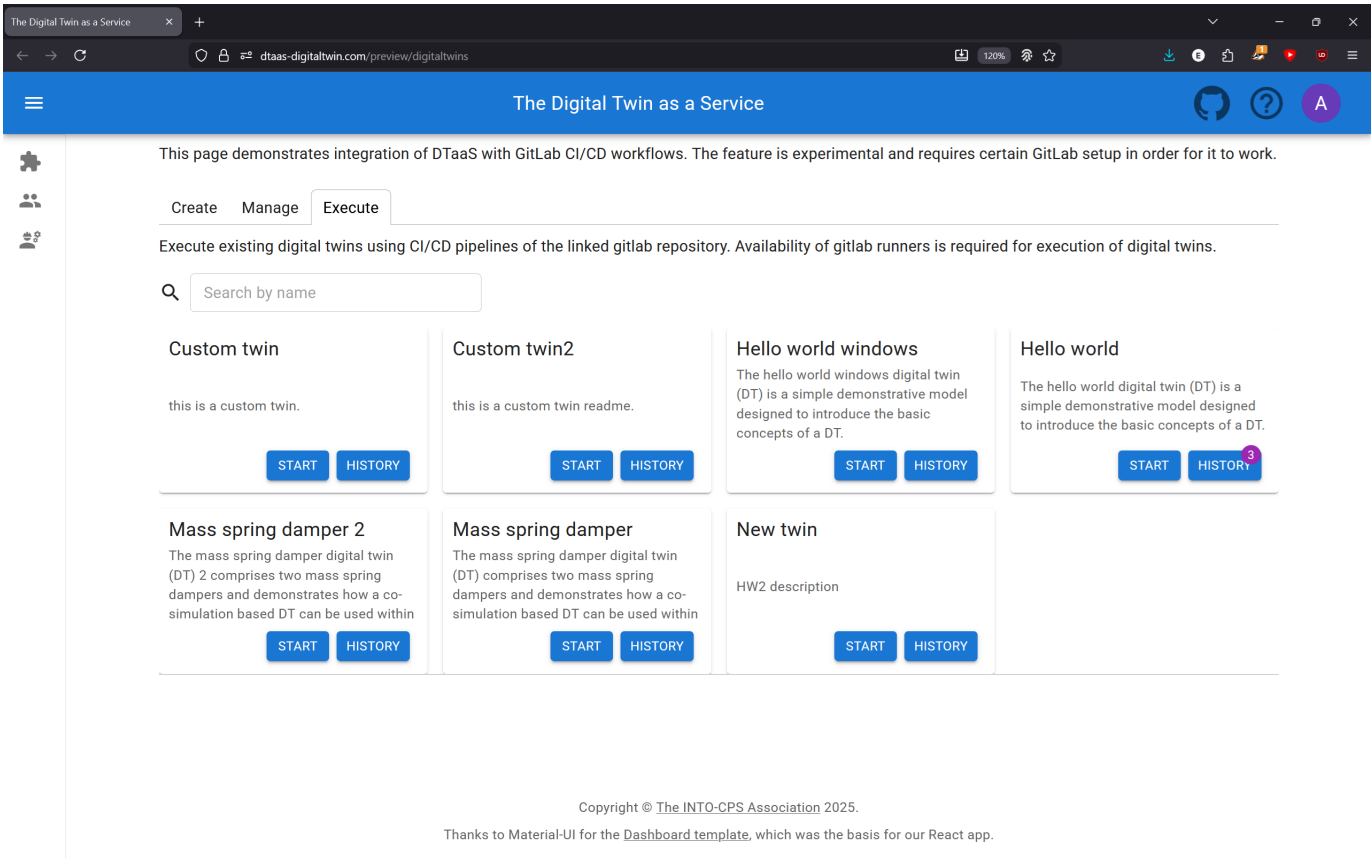
{
  "msd2": {
    "msd2i": {
      "fk": [
        "{msd1}.msd1i.fk"
      ],
      "logVariables": {
        "msd2": {
          "msd2i": [
            "x2", "v2"
          ]
        }
      },
      "parameters": {
        "msd2": {
          "msd2i": {
            "c2": 1.0,
            "cc": 1.0,
            "d2": 1.0,
            "dc": 1.0,
            "m2": 1.0
          }
        }
      },
      "algorithm": {
        "type": "fixed-step",
        "size": 0.001
      },
      "loggingOn": false,
      "overrideLogLevel": "INFO"
    }
  }
}

```

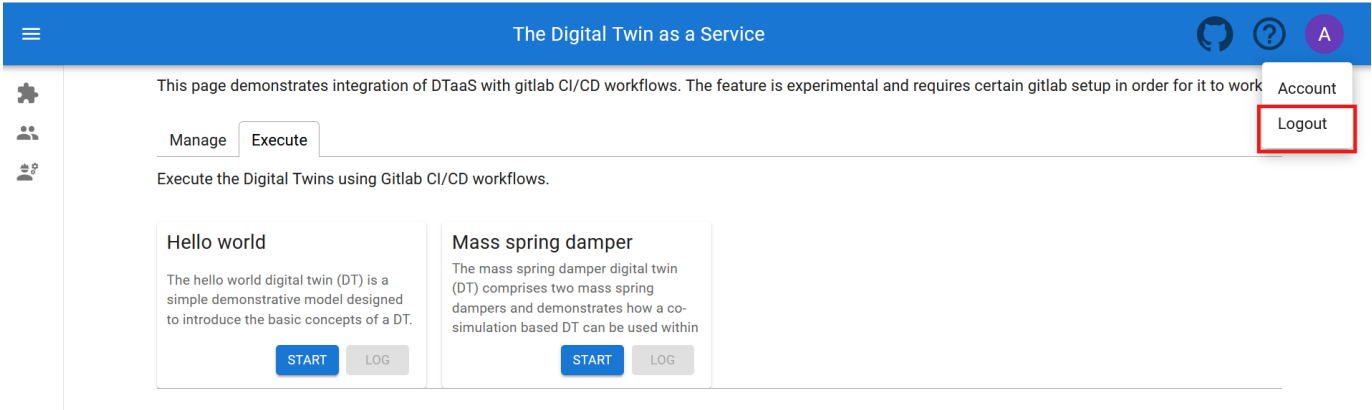
CANCEL SAVE

EXECUTE TAB

Digital Twins can be executed using GitLab CI/CD workflows. Multiple digital twins can be executed simultaneously.



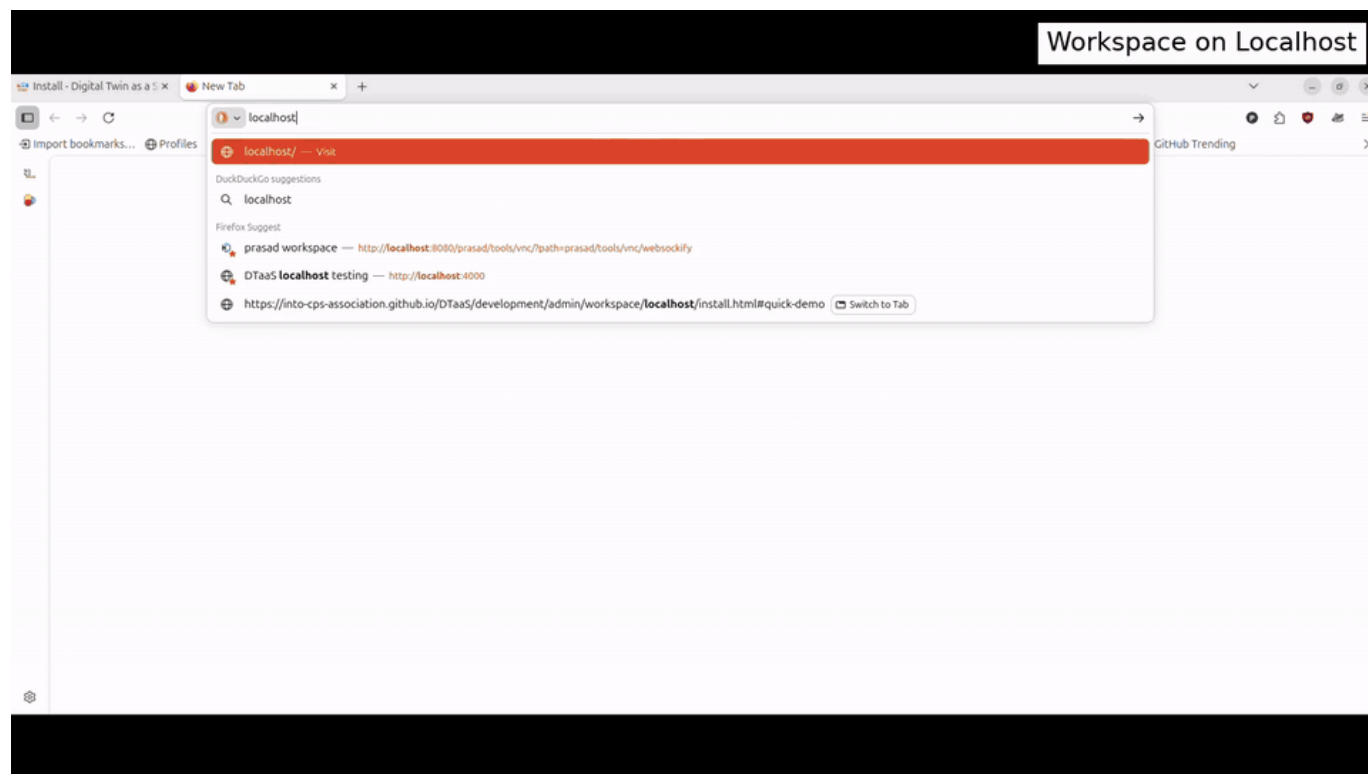
Finally logout



The browser must be closed to completely exit the DTaaS software platform.

3.3.2 Workspace

This page provides a screenshot-driven preview of the website serving the workspace software platform. A quick screen recording is available as well.



The user workspace in DTaaS is a personal Linux environment provided to each registered user. The <https://intocps.org> is used in this page as an illustration for base URL of the installation. Do replace it with the correct URL of the installation.

Visit the Workspace Installation

URL: <https://intocps.org> (specified in `config/client.js` file)

Navigation begins by visiting the website of the DTaaS instance for which the user is registered.

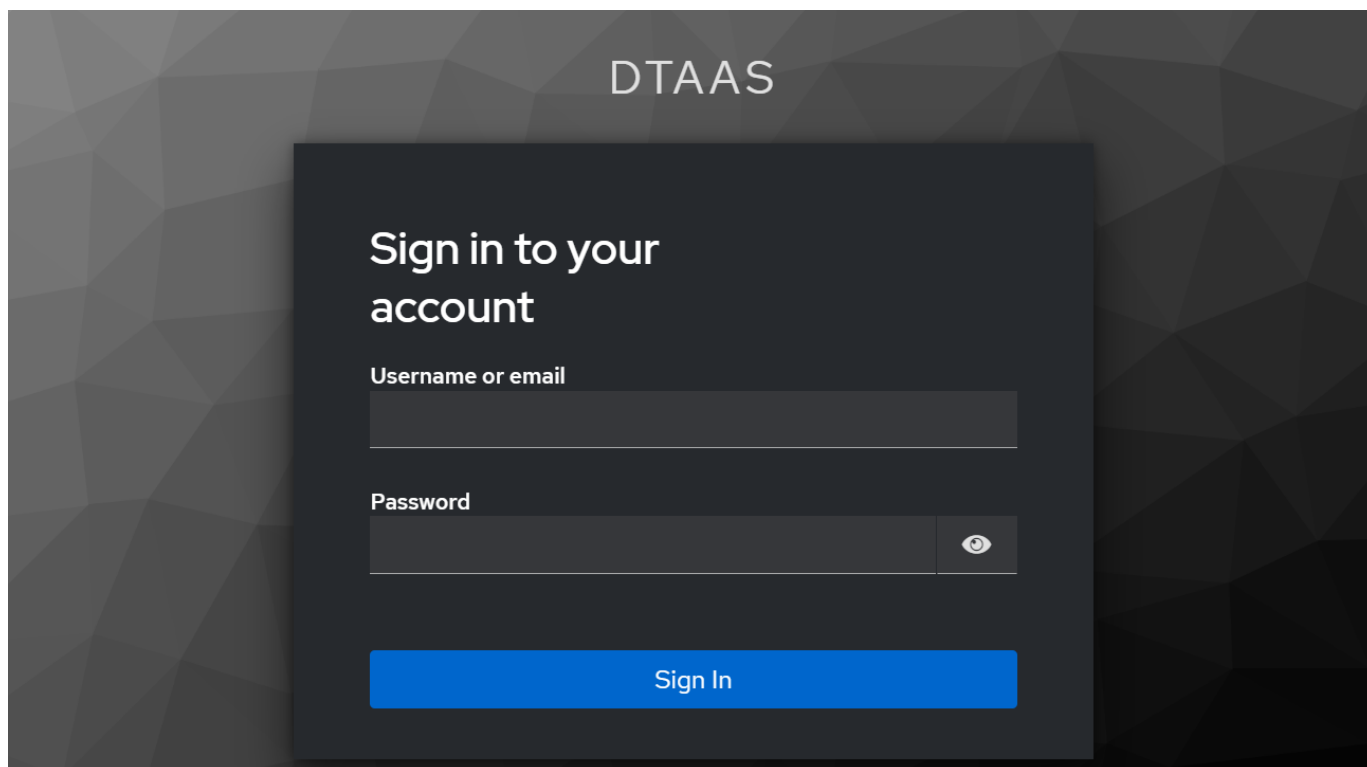


DTaaS Workspace

Redirected to Authorization Provider

URL: `https://intocps.org/auth/realms/dtaas/xxxx` (specified in `config/.env` file)

The browser redirects to the Keycloak Authorization page for the DTaaS Workspace.



The email/username and password should be entered.

The browser redirects to the Application page.

After successful authentication, redirection to the login page of the DTaaS website occurs.

URL: `https://intocps.org`

The DTaaS website employs an additional layer of security - the third-party authorization protocol known as [OIDC](#). This protocol provides secure access to a DTaaS installation for users with active accounts at the selected OIDC service provider. This implementation also uses Keycloak as the OIDC provider.

After successful authorization, redirection to the **Library** page of the DTaaS website occurs.

URL: `https://intocps.org/library/xxxx`

Three icons are located on the top-right of the webpage. The hyperlink on the **?** icon redirects to the help page, while the hyperlink on the **github icon** redirects to the GitHub code repository. The **Account** using the top-right purple **A** icon provides options to [change settings](#) and logout.

Menu Items

The menu is hidden by default. Only the icons of menu items are visible. Clicking on the **three horizontal bars** icon in the top-left corner of the page reveals the menu.

The Digital Twin as a Service

Functions Models Tools Data Digital Twins

The functions responsible for pre- and post-processing of: data inputs, data outputs, control outputs. The data science libraries and functions can be used to create useful function assets for the platform. In some cases, Digital Twin models require calibration prior to their use; functions written by domain experts along with right data inputs can make model calibration an achievable goal. Another use of functions is to process the sensor and actuator data of both Physical Twins and Digital Twins.

Private Common

These reusable assets are only visible to you. Other users can not use these assets in their digital twins.

jupyter

File View Settings Help

Files Running

Select items to perform actions on them.

/ functions /

Name	Modified	Size
------	----------	------

Three menu items are available:

Library: For management of reusable library assets. Files can be uploaded, downloaded, created, and modified on this page.

Digital Twins: For management of digital twins. A Jupyter Lab page is presented from which digital twins can be executed.

Workbench: Not all digital twins can be managed within Jupyter Lab. Additional tools are available on this page.

Library Page

The functions responsible for pre- and post-processing of: data inputs, data outputs, control outputs. The data science libraries and functions can be used to create useful function assets for the platform. In some cases, Digital Twin models require calibration prior to their use; functions written by domain experts along with right data inputs can make model calibration an achievable goal. Another use of functions is to process the sensor and actuator data of both Physical Twins and Digital Twins.

Private Common

These reusable assets are only visible to you. Other users can not use these assets in their digital twins.

jupyter

File View Settings Help

Files Running

Select items to perform actions on them.

/ functions /

☐	Name	Modified	Size
--------------------------	------	----------	------

Five tabs are displayed, each corresponding to one type of digital twin asset. Each tab provides help text to guide users on the asset type.

Functions

The functions responsible for pre- and post-processing of: data inputs, data outputs, control outputs. The data science libraries and functions can be used to create useful function assets for the platform. In some cases, Digital Twin models require calibration prior to their use; functions written by domain experts along with right data inputs can make model calibration an achievable goal. Another use of functions is to process the sensor and actuator data of both Physical Twins and Digital Twins.

Data

The data sources and sinks available to a digital twins. Typical examples of data sources are sensor measurements from Physical Twins, and test data provided by manufacturers for calibration of models. Typical examples of data sinks are visualisation software, external users and data storage services. There exist special outputs such as events, and commands which are akin to control outputs from a Digital Twin. These control outputs usually go to Physical Twins, but they can also go to another Digital Twin.

Models

The model assets are used to describe different aspects of Physical Twins and their environment, at different levels of abstraction. Therefore, it is possible to have multiple models for the same Physical Twin. For example, a flexible robot used in a car production plant may have structural model(s) which will be useful in tracking the wear and tear of parts. The same robot can have a behavioural model(s) describing the safety guarantees provided by the robot manufacturer. The same robot can also have a functional model(s) describing the part manufacturing capabilities of the robot.

Tools

The software tool assets are software used to create, evaluate and analyse models. These tools are executed on top of a computing platforms, i.e., an operating system, or virtual machines like Java virtual machine, or inside docker containers. The tools tend to be platform specific, making them less reusable than models. A tool can be packaged to run on a local or distributed virtual machine environments thus allowing selection of most suitable execution environment for a Digital Twin. Most models require tools to evaluate them in the context of data inputs. There exist cases where executable packages are run as binaries in a computing environment. Each of these packages are a pre-packaged combination of models and tools put together to create a ready to use Digital Twins.

Digital Twins

These are ready to use digital twins created by one or more users. These digital twins can be reconfigured later for specific use cases.

Two sub-tabs exist: **private** and **common**. Library assets in the private category are visible only to the logged-in user, while library assets in the common category are available to all users.

Further explanation on the placement of reusable assets within each type and the underlying directory structure on the server is available on the [assets page](#).

Note

Assets (files) can be uploaded using the **upload** button.

i The file manager is based on Jupyter Notebook, and all tasks available in Jupyter Notebook can be performed here.

Digital Twins Page

URL: <https://intocps.org/digitaltwins>

Copyright © The INTO-CPS Association 2024.

Thanks to Material-UI for the [Dashboard template](#), which was the basis for our React app.

The digital twins page contains three tabs, and the central pane opens Jupyter Lab. The three tabs provide helpful instructions on suggested tasks for the **Create - Execute - Analyze** lifecycle phases of a digital twin. More explanation is available on the [lifecycle phases of digital twin](#).



Create digital twins from tools provided within user workspaces. Each digital twin will have one directory. It is suggested that user provide one bash shell script to run their digital twin. Users can create the required scripts and other files from tools provided in Workbench page.

??? tip Execute Digital twins are executed from within user workspaces. The given bash script gets executed from digital twin directory. Terminal-based digital twins can be executed from VSCode and graphical digital twins can be executed from VNC GUI. The results of execution can be placed in the data directory.



The analysis of digital twins requires running of digital twin script from user workspace. The execution results placed within data directory are processed by analysis scripts and results are placed back in the data directory. These scripts can either be executed from VSCode and graphical results or can be executed from VNC GUI.

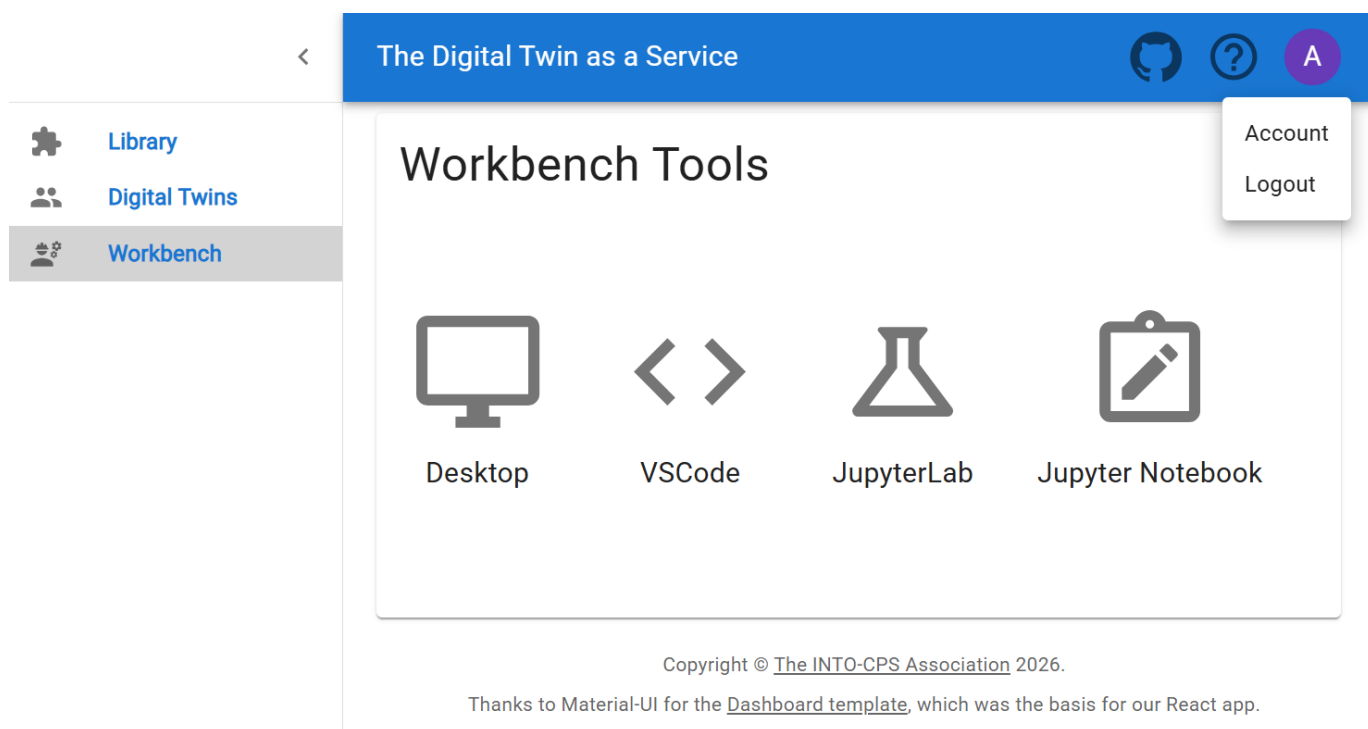
i The reusable assets (files) displayed in the file manager are also available in Jupyter Lab. Additionally, a git plugin is installed in Jupyter Lab that enables linking files with external git repositories.

Workbench

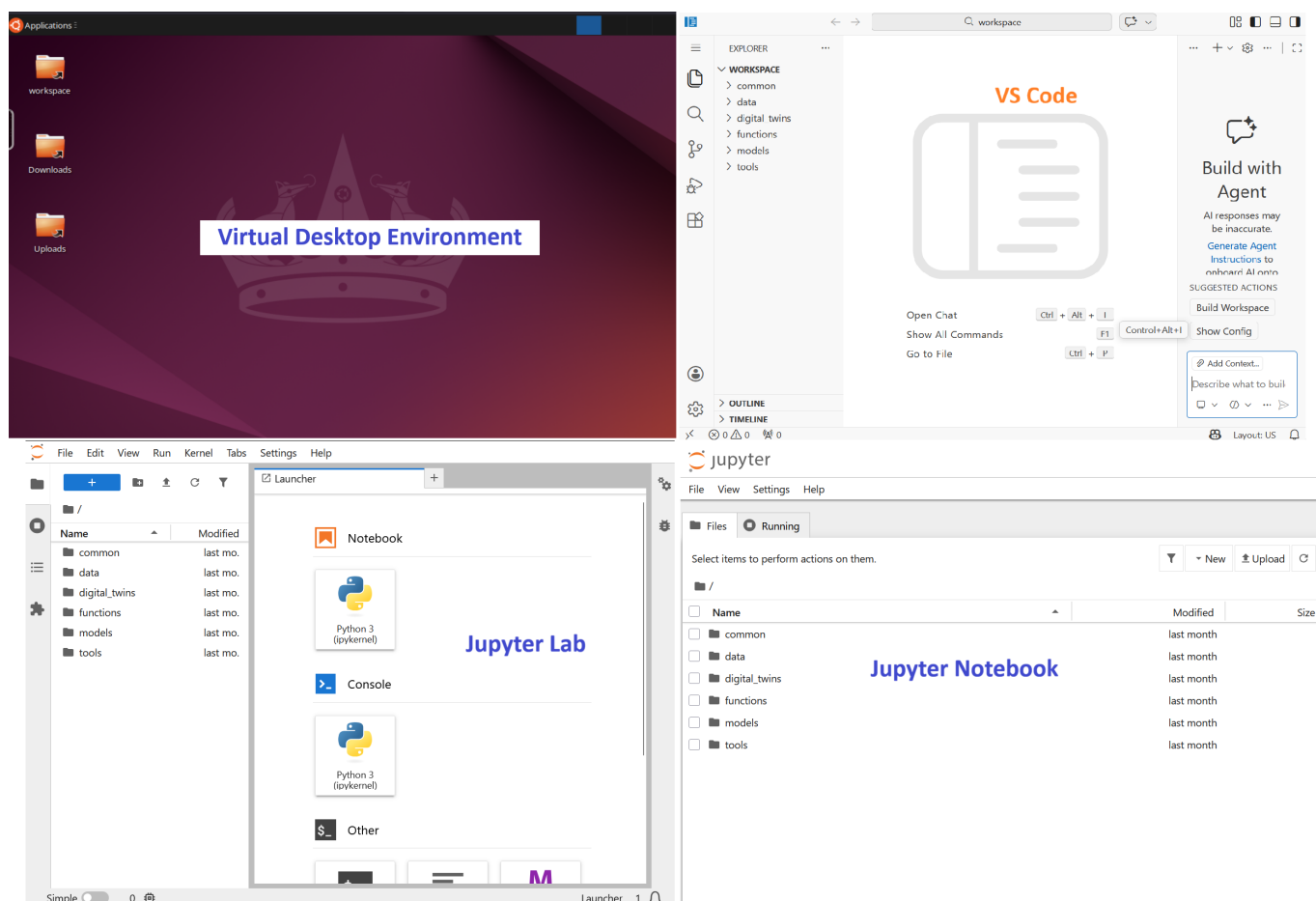
URL: <https://intocps.org/workbench>

The **Workbench** page provides links to four integrated tools:

Tool	Description
Desktop	XFCE-based Linux desktop environment in the browser
VS Code	Browser-based VS Code editor (code-server)
Jupyter Lab	Interactive notebook environment
Jupyter Notebook	Classic Jupyter interface



Screenshots of the pages opened by clicking on first four icons are shown:

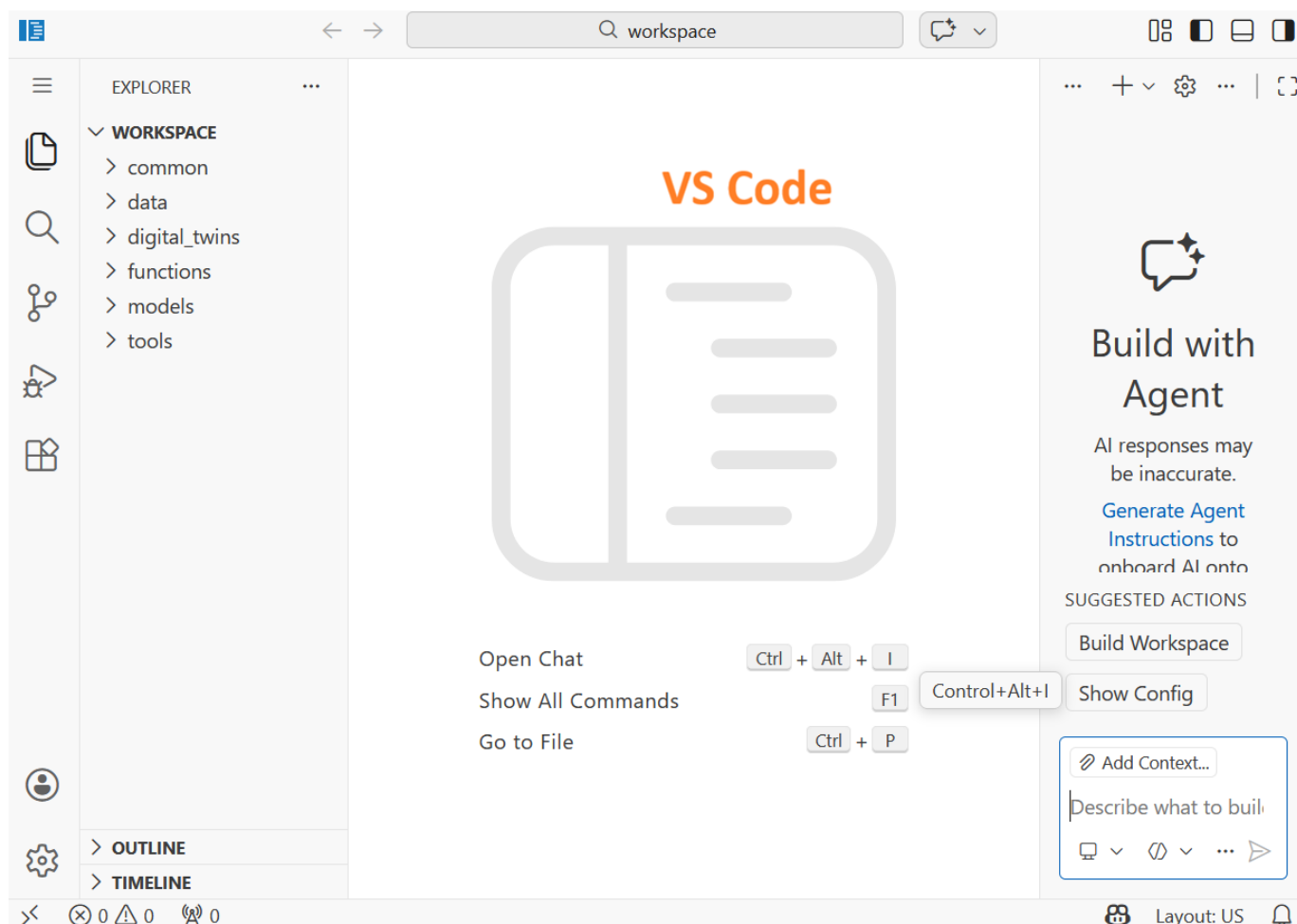


The hyperlinks open in new browser tabs.

VS Code Workspace

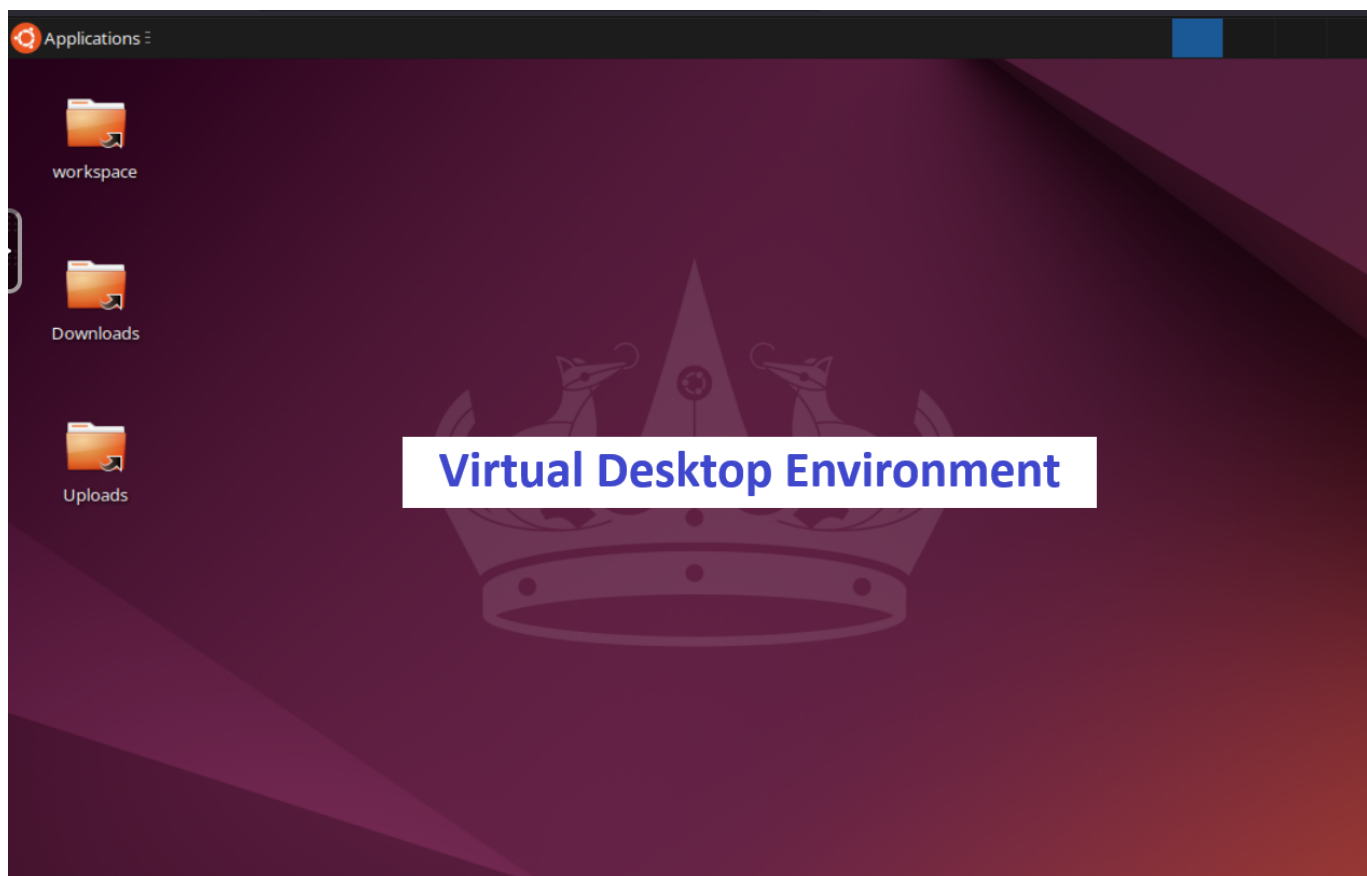
The VS Code interface provides:

- File explorer for navigating the workspace
- Integrated terminal for running commands
- Extensions for Python, Markdown, and other languages



XFCE Desktop

The XFCE desktop provides a full graphical Linux environment accessible from the browser. This is useful for running GUI-based tools that cannot be used from a terminal alone.



Workspace File System

Each user's workspace container mounts two directories:

```

1  /workspace/          # Private user files (read-write)
2  |— data/
3  |— digital_twins/
4  |— functions/
5  |— models/
6  |— tools/
7
8  /workspace/common/   # Shared library assets
9  |— data/
10 |— digital_twins/
11 |— functions/
12 |— models/
13 |— tools/

```

Files in `/workspace/` are private to each user. Files in `/workspace/common/` are shared across all users. Administrators can optionally mount `/workspace/common` as read-only via the compose configuration.

Installing Software

Users can install additional software inside their workspace (XFCE Desktop / Jupyter Lab / VSCode):

```

1  sudo apt-get install <package>
2  pip install <package>

```

Installed software is private to each user's workspace and persists across sessions.

Running Digital Twins

Digital twins stored in `/workspace/digital_twins/` can be executed by running their lifecycle scripts directly in the terminal:

```
1 cd /workspace/digital_twins/<dt-name>
2 lifecycle/create
3 lifecycle/execute
```

Summary

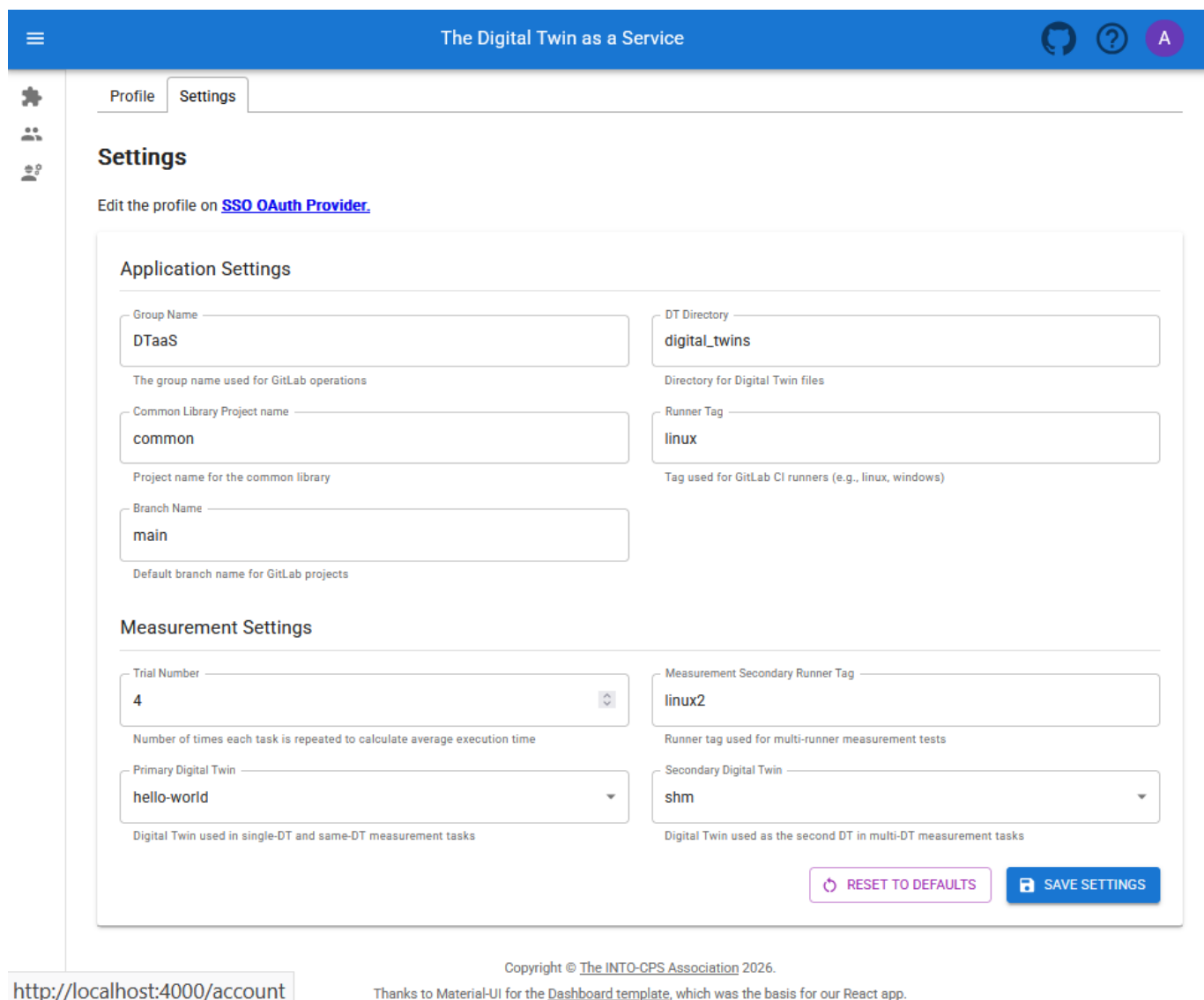
The DTaaS workspace gives each user a private, persistent Linux environment with browser-based access to a desktop, terminal, and development tools. Users can install software, run digital twins, and develop new DT assets within their workspace.

3.3.3 Settings

Settings are important both during initial DTaaS setup and during use when working with different configurations. All the most important settings have been consolidated on one page for both of these scenarios.

Changing Settings

The parameters used for sending and receiving information to the storage and execution services (e.g., GitLab) may need adjustment to match the infrastructure. Navigation to **Account** using the top-right purple **A** icon followed by selection of the **Settings** tab displays the following adjustable parameters:



The screenshot shows the 'Settings' page of 'The Digital Twin as a Service'. The page is divided into two main sections: 'Application Settings' and 'Measurement Settings'. The 'Application Settings' section includes fields for 'Group Name' (DTaaS), 'Common Library Project name' (common), 'Branch Name' (main), 'DT Directory' (digital_twins), and 'Runner Tag' (linux). The 'Measurement Settings' section includes fields for 'Trial Number' (4), 'Measurement Secondary Runner Tag' (linux2), 'Primary Digital Twin' (hello-world), and 'Secondary Digital Twin' (shm). At the bottom of the settings form are two buttons: 'RESET TO DEFAULTS' and 'SAVE SETTINGS'. The footer of the page contains the copyright information 'Copyright © The INTO-CPS Association 2026.' and a URL 'http://localhost:4000/account'.

There are two categories of settings - **Application Settings**, and **Measurement Settings**. Application settings are used for DevOps features while measurement settings are used for benchmarking the DevOps features.

Following is a description of what each parameter does.

APPLICATION SETTINGS

Group Name

The Group Name denotes the highest level of organizational abstraction concerned with on the storage service, namely Groups. A GitLab group is required to use the DTaaS. Within the group, projects reside, which must match the usernames of system

users. More information about the [file organisation](#) is available. This parameter must be set to the case-insensitive name of the group.

Default: DTaaS

Common Library Project name

One project within the group serves as the Digital Twins *Library*. Through the DTaaS, the files inside the Library are accessible to all users and can be copied to individual user projects as needed. This parameter specifies the project name of the Library, and must match that name.

Default: common

DT Directory

Within the common library and user projects, files related to Digital Twins are stored within a designated folder. This is the name chosen for that folder.

Default: Digital_Twins

Branch Name

This parameter determines which branch to search for data (Twins, Functions, etc.) within user and library projects. This parameter also determines which branch's Digital Twins are executed.

Default: master

Runner Tag

The (GitLab) runners responsible for executing Digital Twin code must be associated with a tag. Only one tag can be specified, and it **cannot** be left blank, or the job of running the twin will not be processed. This is a limitation of the DTaaS.

Default: linux

MEASUREMENT SETTINGS

Trial Number

The number of times a test is run during benchmark runs.

Default: 3

Measurement Secondary Runner Tag

Some benchmark runs require two GitLab runners. The *Runner Tag* in **Application Settings** points to first runner, while *Measurement Secondary Runner Tag* points to the second runner.

Default: windows

Primary Digital Twin

The drop-down menu lists all the available digital twins from which the first digital twin is to be selected.

Default: first digital twin in alphabetical order

Secondary Digital Twin

The drop-down menu lists all the available digital twins from which the second digital twin is to be selected.

Default: second digital twin in alphabetical order

Saving and Resetting Changes

When satisfied with the changes, **SAVE SETTINGS** must be pressed for them to persist after leaving the Settings page. If a mistake was made, the settings can be reset to their default values by pressing the **RESET TO DEFAULTS** button. The reset values are saved automatically, so additional saving is not required. The Saving and Resetting buttons on the page:

**RESET TO DEFAULTS****SAVE SETTINGS**

The default values can be found and modified in the code if needed.

Return to the DevOps pages (i.e., Digital Twin Preview) is now possible. **Refreshing ensures fresh data from the remote repository is fetched**, and the Digital Twins should be visible, ready to be executed, edited, and shared.

Summary

This document has described how to edit the settings for initializing the DTaaS to a project and for continuous use (i.e., modifying Runner Tag and Branch). The need to save changes and how to return to default values if a mistake is made have been discussed.

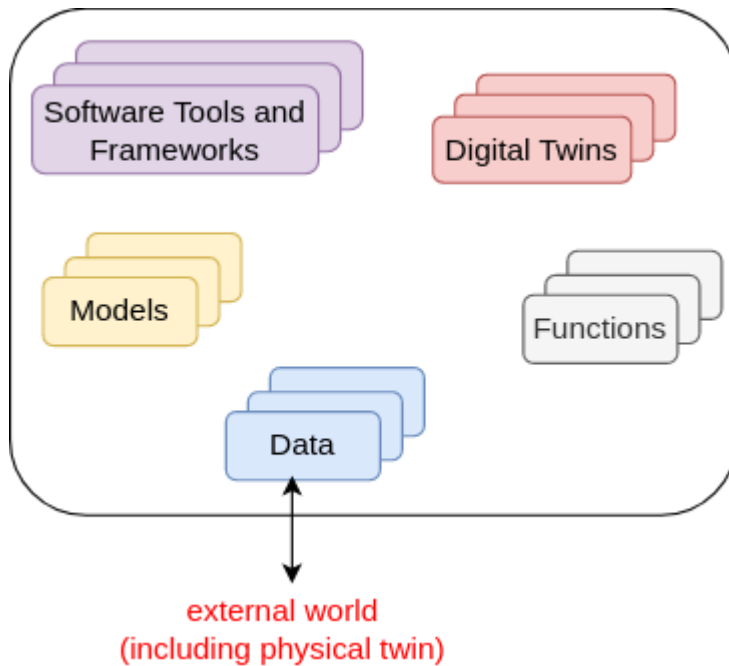
3.4 Reusable Assets

3.4.1 Reusable Assets

The reusability of digital twin assets facilitates efficient work with digital twins. Reusability of assets is a fundamental feature of the platform[1].

Kinds of Reusable Assets

The DTaaS platform categorizes all reusable library assets into six categories:



DATA

The data sources and sinks available to a digital twins. Typical examples of data sources are sensor measurements from Physical Twins, and test data provided by manufacturers for calibration of models. Typical examples of data sinks are visualisation software, external users and data storage services. There exist special outputs such as events, and commands which are akin to control outputs from a Digital Twin. These control outputs usually go to Physical Twins, but they can also go to another Digital Twin.

MODELS

The model assets are used to describe different aspects of Physical Twins and their environment, at different levels of abstraction. Therefore, it is possible to have multiple models for the same Physical Twin. For example, a flexible robot used in a car production plant may have structural model(s) which will be useful in tracking the wear and tear of parts. The same robot can have a behavioural model(s) describing the safety guarantees provided by the robot manufacturer. The same robot can also have a functional model(s) describing the part manufacturing capabilities of the robot.

TOOLS

The software tool assets are software used to create, evaluate and analyse models. These tools are executed on top of a computing platforms, i.e., an operating system, or virtual machines like Java virtual machine, or inside docker containers. The tools tend to be platform specific, making them less reusable than models. A tool can be packaged to run on a local or distributed virtual machine environments thus allowing selection of most suitable execution environment for a Digital Twin. Most models require tools to evaluate them in the context of data inputs. There exist cases where executable packages are run as binaries in a computing environment. Each of these packages are a pre-packaged combination of models and tools put together to create a ready to use Digital Twins.

FUNCTIONS

The functions responsible for pre- and post-processing of: data inputs, data outputs, control outputs. The data science libraries and functions can be used to create useful function assets for the platform. In some cases, Digital Twin models require calibration prior to their use; functions written by domain experts along with right data inputs can make model calibration an achievable goal. Another use of functions is to process the sensor and actuator data of both Physical Twins and Digital Twins.

DIGITAL TWINS

These are ready to use digital twins created by one or more users. These digital twins can be reconfigured later for specific use cases.

File System Structure

Each user has their assets put into five different directories named above. In addition, there will also be common library assets that all users have access to. A simplified example of the structure is as follows:

```

1 workspace/
2   data/
3     data1/ (ex: sensor)
4       filename (ex: sensor.csv)
5       README.md
6     data2/ (ex: turbine)
7       README.md (remote source; no local file)
8     ...
9   digital_twins/
10    digital_twin-1/ (ex: incubator)
11      config (yaml and json)
12      README.md (usage instructions)
13      description.md (short summary of digital twin)
14      lifecycle/ (directory containing lifecycle scripts)
15    digital_twin-2/ (ex: mass spring damper)
16      config (yaml and json)
17      README.md (usage instructions)
18      description.md (short summary of digital twin)
19      lifecycle/ (directory containing lifecycle scripts)
20    digital_twin-3/ (ex: model swap)
21      config (yaml and json)
22      README.md (usage instructions)
23      description.md (short summary of digital twin)
24      lifecycle/ (directory containing lifecycle scripts)
25    ...
26  functions/
27    function1/ (ex: graphs)
28      filename (ex: graphs.py)
29      README.md
30    function2/ (ex: statistics)
31      filename (ex: statistics.py)
32      README.md
33    ...
34  models/
35    model1/ (ex: spring)
36      filename (ex: spring.fmu)
37      README.md
38    model2/ (ex: building)
39      filename (ex: building.skp)
40      README.md
41    model3/ (ex: rabbitmq)
42      filename (ex: rabbitmq.fmu)
43      README.md
44    ...
45  tools/
46    tool1/ (ex: maestro)
47      filename (ex: maestro.jar)
48      README.md
49    ...
50  common/
51    data/
52    functions/
53    models/
54    tools/

```

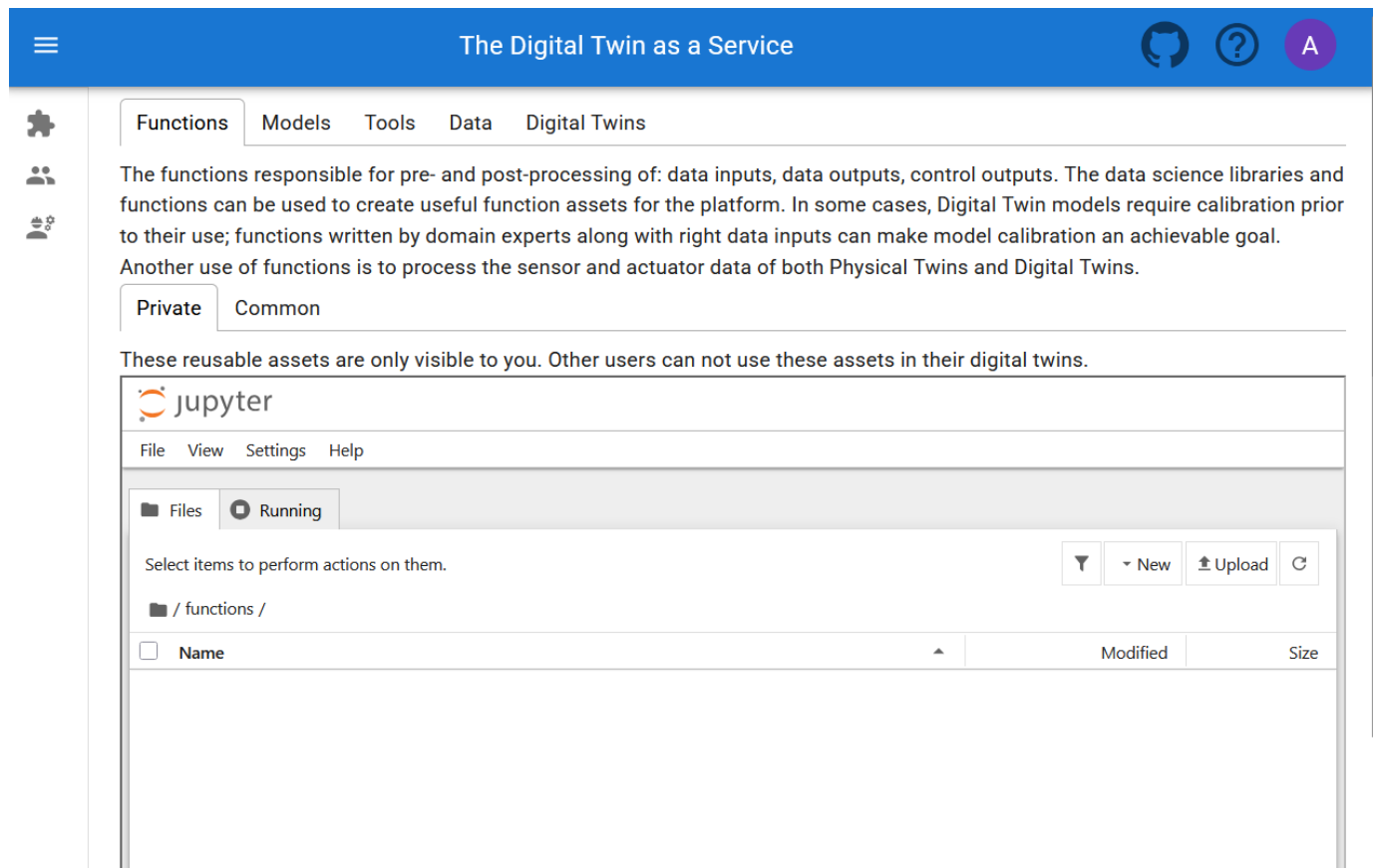


The DTaaS is agnostic to the format of assets. The only requirement is that they are files which can be uploaded on the Library page. Directories can be compressed as single files and uploaded. The files can be decompressed into directories from a Terminal or xfce Desktop available on the Workbench page.

A recommended file system structure for storing assets is also available in [DTaaS examples](#).

Upload Assets

Users can upload assets into their workspace using the Library page of the website.



Navigation into a directory followed by clicking on the **upload** button allows uploading files or directories into the workspace. These assets then become available in all workbench tools. New assets can also be created on the page by clicking on the **new** dropdown menu. This simple web interface allows creation of text-based files. Other files must be uploaded using the **upload** button.

The user workbench provides the following services:

- Jupyter Notebook and Lab
- VS Code
- XFCE Desktop Environment available via VNC
- Terminal

Users can also bring DT assets into user workspaces from external sources using any of the above-mentioned services. Developers using *git* repositories can clone from and push to remote git servers. Users can also use widely-used file transfer protocols such as FTP and SCP to bring the required DT assets into the user workspaces.

References

- [1]: Talasila, Prasad, et al. "Composable digital twins on Digital Twin as a Service platform." *Simulation* 101.3 (2025): 287-311.

3.4.2 Library Microservice

The lib microservice is responsible for handling and serving the contents of library assets of the DTaaS platform. It provides API endpoints for clients to query, and fetch these assets.

This document provides instructions for using the library microservice.

The [assets](#) page describes suggested storage conventions for library assets.

Once assets are stored in the library, they become available in the user workspace.

Application Programming Interface (API)

The lib microservice application provides services at two end points:

GraphQL API Endpoint: `http://intocps.org/lib`

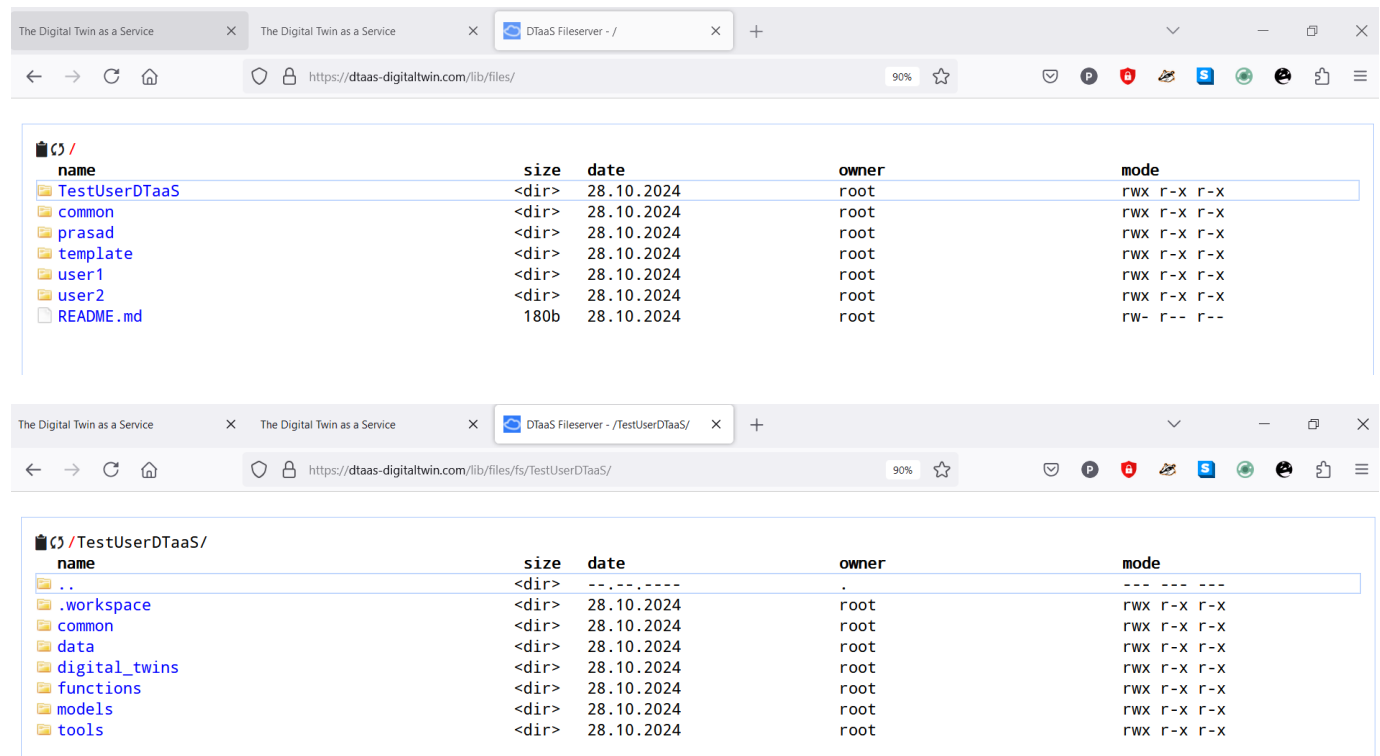
HTTP Endpoint: `http://intocps.org/lib/files`

HTTP PROTOCOL

Endpoint: `localhost:PORT/lib/files`

This option needs to be enabled with `-H http.json` flag. The regular file upload and download options become available.

Here are sample screenshots.



GraphQL PROTOCOL

Endpoint: `localhost:PORT/lib`

The `http://intocps.org/lib` URL opens a GraphQL playground.

The query schema can be examined and sample queries can be tested there. GraphQL queries must be sent as HTTP POST requests to receive responses.

The library microservice provides two API calls:

- Provide a list of contents for a directory
- Fetch a file from the available files

The API calls are accepted over GraphQL and HTTP API endpoints. The format of accepted queries is:

PROVIDE LIST OF CONTENTS FOR A DIRECTORY

To retrieve a list of files in a directory, use the following GraphQL query.

Replace `path` with the desired directory path.

send requests to: <https://intocps.org/lib>

GraphQL query for list of contents

```
1 query {  
2   listDirectory(path: "user1") {  
3     repository {  
4       tree {  
5         blobs {  
6           edges {  
7             node {  
8               name  
9               type  
10            }  
11          }  
12        }  
13        trees {  
14          edges {  
15            node {  
16              name  
17              type  
18            }  
19          }  
20        }  
21      }  
22    }  
23  }  
24 }
```

GraphQL response for list of contents

```

1  {
2    "data": {
3      "listDirectory": {
4        "repository": {
5          "tree": {
6            "blobs": {
7              "edges": []
8            },
9            "trees": {
10             "edges": [
11               {
12                 "node": {
13                   "name": "common",
14                   "type": "tree"
15                 }
16               },
17               {
18                 "node": {
19                   "name": "data",
20                   "type": "tree"
21                 }
22               },
23               {
24                 "node": {
25                   "name": "digital twins",
26                   "type": "tree"
27                 }
28               },
29               {
30                 "node": {
31                   "name": "functions",
32                   "type": "tree"
33                 }
34               },
35               {
36                 "node": {
37                   "name": "models",
38                   "type": "tree"
39                 }
40               },
41               {
42                 "node": {
43                   "name": "tools",
44                   "type": "tree"
45                 }
46               }
47             ]
48           }
49         }
50       }
51     }
52   }
53 }

```

HTTP request for list of contents

```

1  POST /lib HTTP/1.1
2  Host: intocps.org
3  Content-Type: application/json
4  Content-Length: 388
5
6  {
7    "query": "query { \n  listDirectory(path: \"user1\") { \n    repository { \n      tree { \n        blobs { \n          edges { \n            node { \n              name \n              type \n            } \n          } \n        } \n      } \n    } \n  } \n}"
8  }

```

HTTP response for list of contents

```

1  HTTP/1.1 200 OK
2  Access-Control-Allow-Origin: *
3  Connection: close
4  Content-Length: 306
5  Content-Type: application/json; charset=utf-8
6  Date: Tue, 26 Sep 2023 20:26:49 GMT
7  X-Powered-By: Express
8  {"data":{"listDirectory":{"repository":{"tree":{"blobs":{"edges":[]},"trees":{"edges":[{"node":{"name":"data","type":"tree"}},{node":{"name":"digital twins","type":"tree"}},{node":{"name":"functions","type":"tree"}},{node":{"name":"models","type":"tree"}},{node":{"name":"tools","type":"tree"}}]}}}}}

```

FETCH A FILE FROM THE AVAILABLE FILES

This query receives directory path and send the file contents to user in response.

To check this query, create a file `files/user2/data/welcome.txt` with content of `hello world`.

GraphQL query for fetch a file

```

1 query {
2   readFile(path: "user2/data/sample.txt") {
3     repository {
4       blobs {
5         nodes {
6           name
7           rawBlob
8           rawTextBlob
9         }
10      }
11    }
12  }
13 }
```

GraphQL response for fetch a file

```

1 {
2   "data": {
3     "readFile": {
4       "repository": {
5         "blobs": {
6           "nodes": [
7             {
8               "name": "sample.txt",
9               "rawBlob": "hello world",
10              "rawTextBlob": "hello world"
11            }
12          ]
13        }
14      }
15    }
16  }
17 }
```

HTTP request for fetch a file

```

1 POST /lib HTTP/1.1
2 Host: intocps.org
3 Content-Type: application/json
4 Content-Length: 217
5 {
6   "query": "query { \n  readFile(path: \"user2/data/welcome.txt\") { \n    repository { \n      blobs { \n        nodes { \n          name \n          rawBlob \n          rawTextBlob \n        } \n      } \n    } \n  } \n}"
7 }
```

HTTP response for fetch a file

```

1 HTTP/1.1 200 OK
2 Access-Control-Allow-Origin: *
3 Connection: close
4 Content-Length: 134
5 Content-Type: application/json; charset=utf-8
6 Date: Wed, 27 Sep 2023 09:17:18 GMT
7 X-Powered-By: Express
8 {"data":{"readFile":{"repository":{"blobs":{"nodes":[{"name":"welcome.txt","rawBlob":"hello world","rawTextBlob":"hello world"}]}}}}}
```

The *path* refers to the file path to look at: For example, *user1* looks at files of **user1**; *user1/functions* looks at contents of *functions/* directory.

3.4.3 Reusable Assets and DevOps

DevOps has been a well established software development practice. An experimental feature integrating DevOps is being introduced in the DTaaS platform.

This feature requires specific installation setup.

- 1. [Integrated GitLab installation](#)
- 2. A valid GitLab repository for the logged in user. See an [example repository](#). This repository can be cloned and customised as needed.
- 3. [A linked GitLab Runner](#) to the user GitLab repository.

Once these requirements are satisfied, the **Library** page shows all the reusable assets available stored in the linked GitLab repository. An empty list is shown if there are no assets of a specific category.

The Digital Twin as a Service

Functions

Models

Tools

Data

Digital Twins

The functions responsible for pre- and post-processing of: data inputs, data outputs, control outputs. The data science libraries and functions can be used to create useful function assets for the platform. In some cases, Digital Twin models require calibration prior to their use; functions written by domain experts along with right data inputs can make model calibration an achievable goal. Another use of functions is to process the sensor and actuator data of both Physical Twins and Digital Twins.

Private

Common

These reusable assets are only visible to you. Other users can not use these assets in their digital twins.

Selection

CLEAR

PROCEED

The page gets populated with any existing assets. All available DTs are shown in the following figure.

The Digital Twin as a Service

FunctionsModelsToolsDataDigital Twins

These are ready to use digital twins created by one or more users. These digital twins can be reconfigured later for specific use cases.

PrivateCommon

These reusable assets are only visible to you. Other users can not use these assets in their digital twins.

Q

Search by name

Hello world

The hello world digital twin (DT) is a simple demonstrative model designed to introduce the basic concepts of a DT.

DETAILSADD

Mass spring damper

The mass spring damper digital twin (DT) comprises two mass spring dampers and demonstrates how a co-

DETAILSREMOVE

Runner

Use DT Runner to execute commands.

DETAILSADD

Shm devops

This project creates a fully-functional demo of CP-SENS project.

DETAILSADD

Yafem demo

Execute YaFEM demo in devops pipeline of DTaaS.

DETAILSADD

Selection

- digital_twins/mass-spring-damper

CLEARPROCEED

Any existing DT asset can be selected and it gets added to the **Selection** pane on the right. After selecting all the required assets, clicking on the *Proceed* button transitions to [DT create stage](#)

- 47/138 -

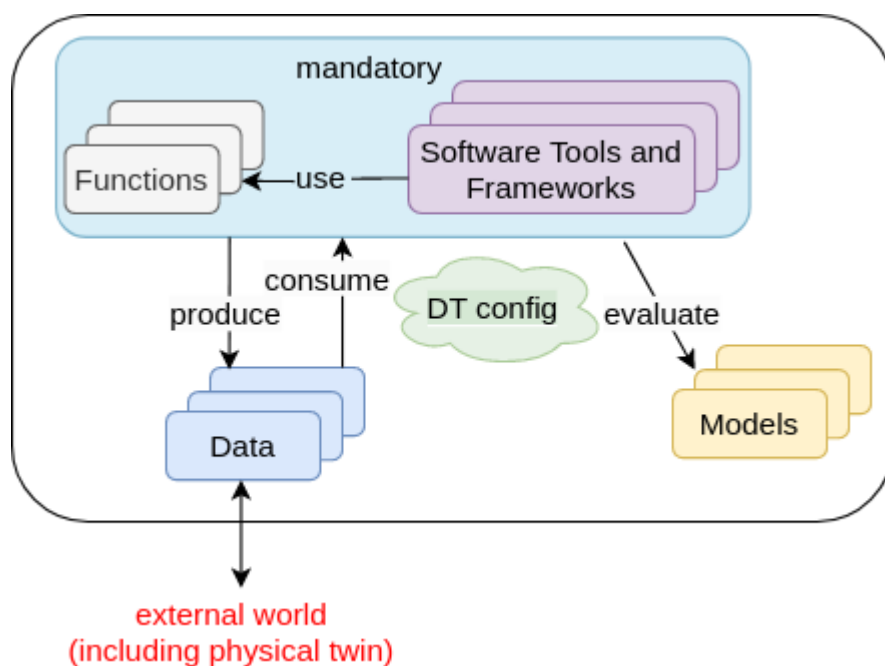
Copyright © 2022 - 2026 The INTO-CPS Association

3.5 Digital Twins

3.5.1 Create a Digital Twin

The first step in digital twin creation involves utilizing the available assets within the workspace. For assets or files residing on a local computer that need to be accessible in the DTaaS workspace, the instructions provided in [library assets](#) should be followed.

Dependencies exist among the library assets. These dependencies are illustrated below.



A digital twin can only be created by linking assets in a meaningful way. This relationship can be expressed using the following mathematical equation:

where D denotes data, M denotes models, F denotes functions, T denotes tools, T denotes DT configuration, and DT is a symbolic notation for a digital twin itself. The expression $DT = D \circ M \circ T \circ F$ denotes composition of a DT from D , M , T , and F assets. The $*$ indicates zero or more instances of an asset, and $+$ indicates one or more instances of an asset.

The DT configuration specifies the relevant assets to use and the potential parameters to be set for these assets. When a DT requires RabbitMQ, InfluxDB, or similar services supported by the platform, the DT configuration must include access credentials for these services.

This generic DT definition is based on DT examples observed in practice. Deviation from this definition is permissible. The only requirement is the ability to execute the DT from either the command line or a graphical desktop environment.



For users new to Digital Twins who may not have distinct digital twin assets but rather a single directory containing all components, it is recommended to upload this monolithic digital twin into the **digital_twin/your_digital_twin_name** directory.

Example

The [Examples](#) repository contains a co-simulation setup for a mass-spring-damper system. This example demonstrates the application of co-simulation techniques for digital twins.

The file system contents for this example are:

```

1 workspace/
2   data/
3     mass-spring-damper
4       input/
5       output/
6
7   digital_twins/
8     mass-spring-damper/
9       cosim.json
10      time.json
11      lifecycle/
12        analyze
13        clean
14        evolve
15        execute
16        save
17        terminate
18      README.md
19
20   functions/
21   models/
22     MassSpringDamper1.fmu
23     MassSpringDamper2.fmu
24
25   tools/
26     common/
27       data/
28       functions/
29       models/
30       tools/
31     maestro-2.3.0-jar-with-dependencies.jar

```

The `workspace/data/mass-spring-damper/` directory contains `input` and `output` data for the mass-spring-damper digital twin.

The two FMU models required for this digital twin are located in the `models/` directory.

The co-simulation digital twin requires the Maestro co-simulation orchestrator. As this is a reusable asset for all co-simulation-based DTs, the tool has been placed in the `common/tools/` directory.

The digital twin configuration is specified in the `digital twins/mass-spring-damper` directory. The co-simulation configuration is defined in two JSON files: `cosim.json` and `time.json`. Documentation for the digital twin can be placed in `digital twins/mass-spring-damper/document.md`.

The launch program for this digital twin is located in `digital twins/mass-spring-damper/lifecycle/execute`. This launch program executes the co-simulation digital twin, which runs until completion and then terminates. The programs in `digital twins/mass-spring-damper/lifecycle` are responsible for lifecycle management of this digital twin. The [lifecycle page](#) provides further explanation of these programs.

Execution of a Digital Twin

A frequent question arises on the run time characteristics of a digital twin. The natural intuition is to say that a digital twin must operate as long as its physical twin is in operation. **If a digital twin runs for a finite time and then ends, can it be called a digital twin? The answer is a resounding YES.** The Industry 4.0 usecases seen among SMEs have digital twins that run for a finite time. These digital twins are often run at the discretion of the user.

Execution of this digital twin involves the following steps:

1. Navigate to the Workbench tools page of the DTaaS website and open VNC Desktop. This opens a new tab in the browser.
2. A page with VNC Desktop and a connect button is displayed. Click on Connect to establish a connection to the Linux Desktop of the workspace.
3. Open a Terminal (the black rectangular icon in the top left region of the tab) and enter the following commands.
4. Download the example files by following the instructions provided in the [examples overview](#).
5. Navigate to the digital twin directory and execute:

```

1 cd /workspace/examples/digital_twins/mass-spring-damper
2 lifecycle/execute

```

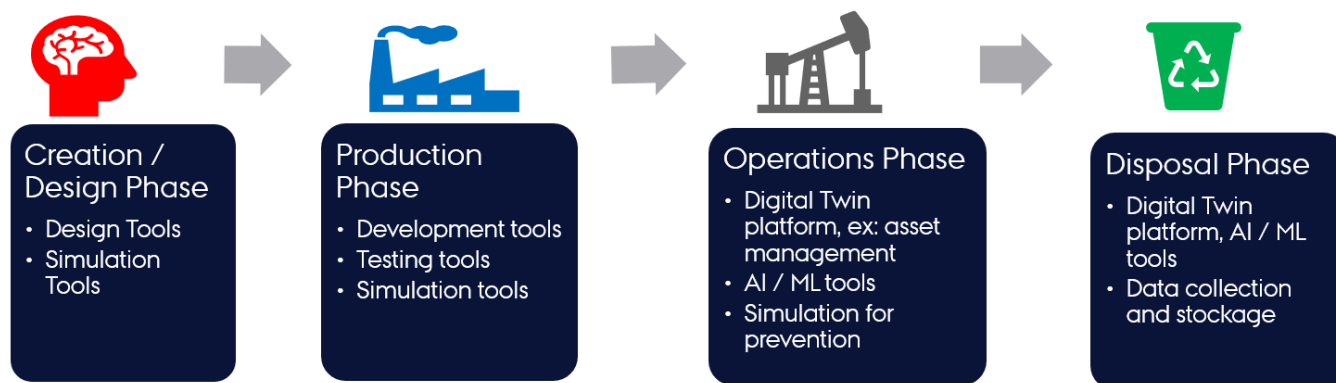
The final command executes the mass-spring-damper digital twin and stores the co-simulation output in `data/mass-spring-damper/output`.

3.5.2 🌱 Digital Twin Lifecycle

Physical products in the real world undergo a product lifecycle. A simplified four-stage product lifecycle is illustrated here.

A digital twin tracking physical products (twins) must evolve in conjunction with the corresponding physical twin.

The possible activities undertaken in each lifecycle phase are illustrated in the figure.



(Ref: Minerva, R, Lee, GM and Crespi, N (2020) Digital Twin in the IoT context: a survey on technical features, scenarios and architectural models. Proceedings of the IEEE, 108 (10). pp. 1785-1824. ISSN 0018-9219.)

Lifecycle Phases

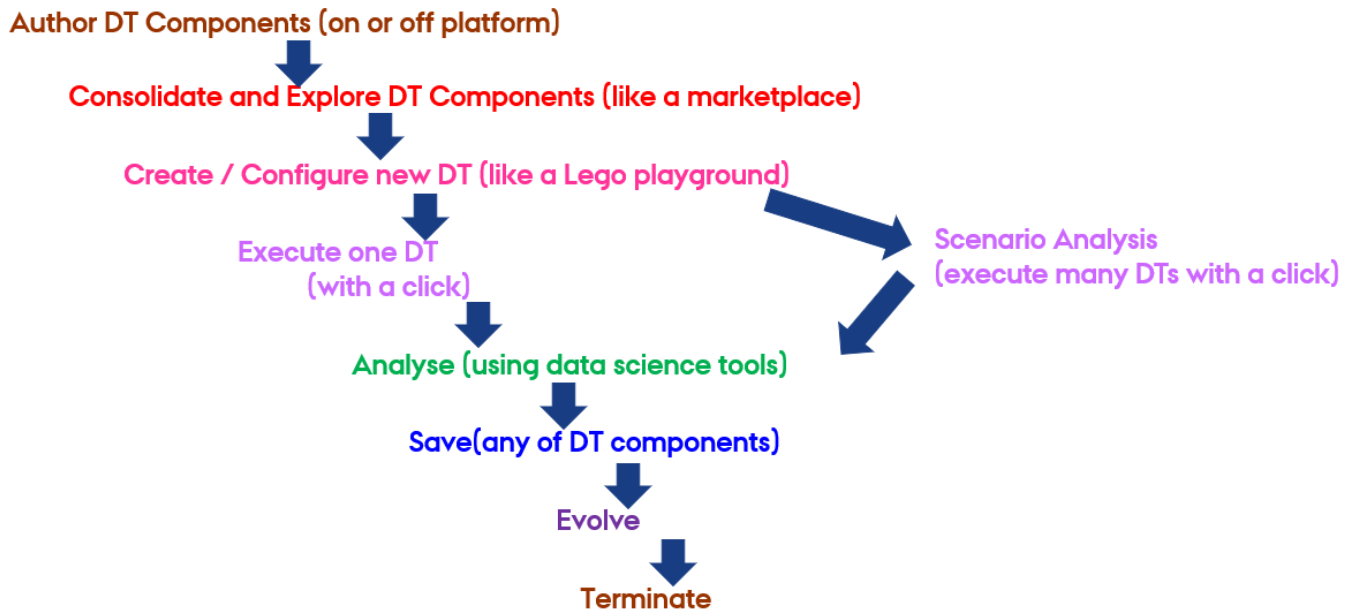
The four-phase lifecycle has been extended to a lifecycle with eight phases. The new phase names and the typical activities undertaken in each phase are outlined in this section[1].

A DT lifecycle consists of **explore**, **create**, **execute**, **save**, **analyse**, **evolve** and **terminate** phases.

Phase	Main Activities
explore	Selection of suitable assets based on user requirements and verification of their compatibility for DT creation.
create	Specification of DT configuration. For existing DTs, no creation phase is required at the time of reuse.
execute	Automated or manual execution of a DT based on its configuration. The DT configuration must be verified before starting execution.
analyse	Examination of DT outputs and decision-making. Outputs may include text files or visual dashboards.
evolve	Reconfiguration of DT primarily based on analysis results.
save	Preservation of DT state to enable future recovery.
terminate	Cessation of DT execution.

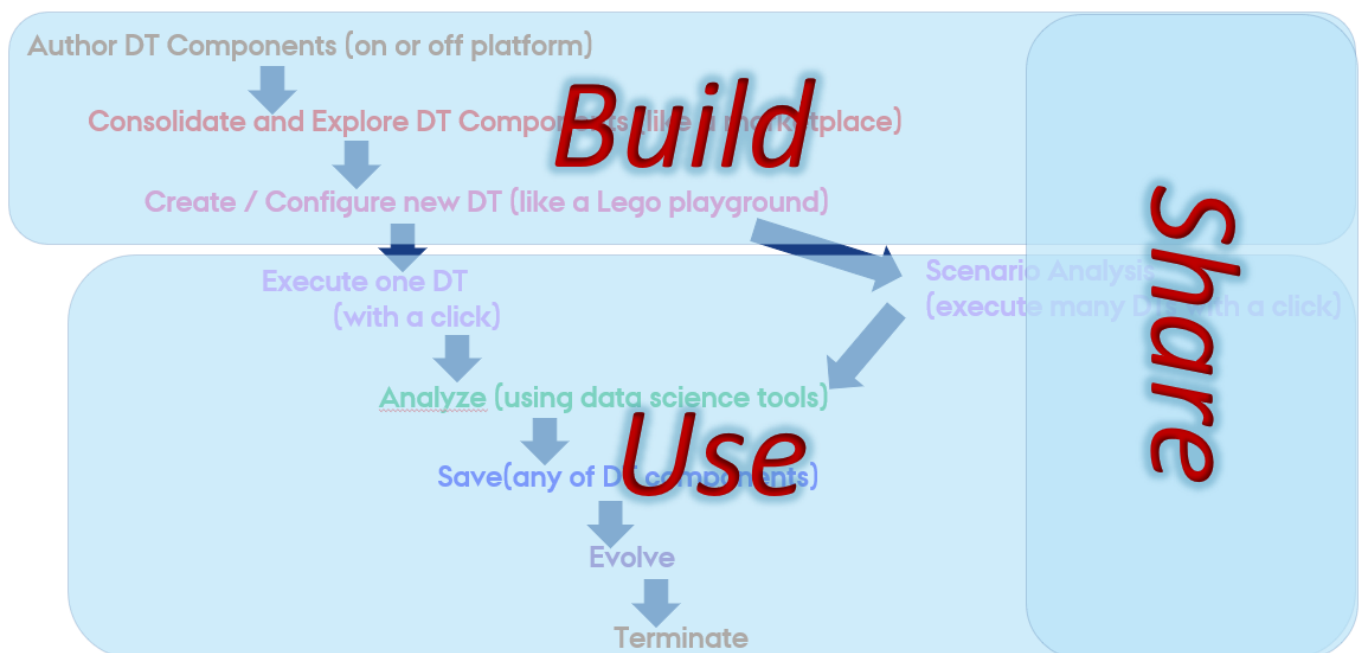
A digital twin faithfully tracking the physical twin lifecycle must support all the phases. Digital twin engineers may also add additional phases to their implementations. Consequently, the DTaaS platform is designed to accommodate the needs of diverse DTs.

A potential linear representation of the tasks undertaken in a digital twin lifecycle is shown here.



This representation shows only one possible pathway. The sequence of steps may be altered as needed.

It is possible to map the lifecycle phases to the **Build-Use-Share** approach of the DTaaS platform.



Although not mandatory, maintaining a matching code structure facilitates DT creation and management within the DTaaS platform. The following structure is recommended:

```

1 workspace/
2   digital_twins/
3     digital-twin-1/
4       lifecycle/
5         analyze
6         clean
7         evolve
8         execute
9         save
10        terminate
  
```

A dedicated program exists for each phase of the DT lifecycle. Each program can be as simple as a script that launches other programs or sends messages to a live digital twin.

i The recommended approach for implementing lifecycle phases within DTaaS is to create scripts. These scripts can be implemented as shell scripts.

Example Lifecycle Scripts

The following example programs/scripts demonstrate management of three phases in the lifecycle of the **mass-spring-damper DT**.

lifecycle/execute

```
1 #!/bin/bash
2 mkdir -p /workspace/data/mass-spring-damper/output
3 #cd ..
4 java -jar /workspace/common/tools/maestro-2.3.0-jar-with-dependencies.jar \
5     import -output /workspace/data/mass-spring-damper/output \
6     --dump-intermediate sg1 cosim.json time.json -i -vi FMI2 \
7     output-dir>debug.log 2>&1
```

The execute phase utilizes the DT configuration, FMU models, and Maestro tool to execute the digital twin. The script also stores the output of co-simulation in `/workspace/data/mass-spring-damper/output`.

A DT may not support a specific lifecycle phase. This intention can be expressed with an empty script and a helpful message if deemed necessary.

lifecycle/analyze

```
1 #!/bin/bash
2 printf "operation is not supported on this digital twin"
```

The lifecycle programs can invoke other programs in the codebase. In the case of the `lifecycle/terminate` program, it calls another script to perform the necessary operations.

lifecycle/terminate

```
1 #!/bin/bash
2 lifecycle/clean
```

References

[1]: Talasila, Prasad, et al. "Composable digital twins on Digital Twin as a Service platform." *Simulation* 101.3 (2025): 287-311.

3.5.3 DevOps

Digital Twin File Structure in GitLab

GitLab is used as a file store for performing DevOps on Digital Twins. The [user interface page](#) is a front-end for this gitlab-backed file storage.

Each DTaaS installation comes with an integrated GitLab. There must be a GitLab group named **dtaas** and a GitLab repository for each user where repository name matches the username. For example, if there are two users, namely *user1* and *user2* on a DTaaS installation, then the following repositories must exist on the linked GitLab installation.

```
1 https://intocps.org/gitlab/dtaas/common.git
2 https://intocps.org/gitlab/dtaas/user1.git
3 https://intocps.org/gitlab/dtaas/user2.git
```

warning

The assets being displayed on the Library preview page come from the `master` branch of the backing GitLab project. Please create a branch named `master` and make it the default branch. This must be done for all the user repositories including the common repository.

Each user repository must also have a specific structure. The required structure is as follows.

```
1 <username>/
2 |─ common/
3 |─ data/
4 |─ digital_twins/
5 |─ functions/
6 |─ models/
7 |─ tools/
8 |─ .gitlab-ci.yml
9 |─ README.md
```

This file structure follows the same pattern user sees on the existing **Library** page.

DIGITAL TWIN STRUCTURE

The `digital_twins` folder contains DTs that have been pre-built by one or more users. The intention is that they should be sufficiently flexible to be reconfigured as required for specific use cases.

Let us look at an example of such a configuration. The [dtaas/user1 repository on gitlab.com](#) contains the `digital_twins` directory with a `hello_world` example. Its file structure looks like this:

```
1 hello_world/
2 |─ lifecycle/ (at least one lifecycle script)
3 | |─ clean
4 | |─ create
5 | |─ execute
6 | |─ terminate
7 |─ .gitlab-ci.yml (GitLab DevOps config for executing lifecycle scripts)
8 |─ description.md (optional but is recommended)
9 |─ README.md (optional but is recommended)
```

The `lifecycle` directory here contains four files - `clean`, `create`, `execute` and `terminate`, which are simple **BASH** scripts. These correspond to stages in a digital twin's lifecycle. Further explanation of digital twin is available on [lifecycle stages](#).

Digital Twins and DevOps

DevOps has been a well established software development practice. An experimental feature integrating DevOps is being introduced in the DTaaS.

The latest workflow recordings are available here:

- [DTaaS DevOps workflows](#)
- [Automation layer demo](#)

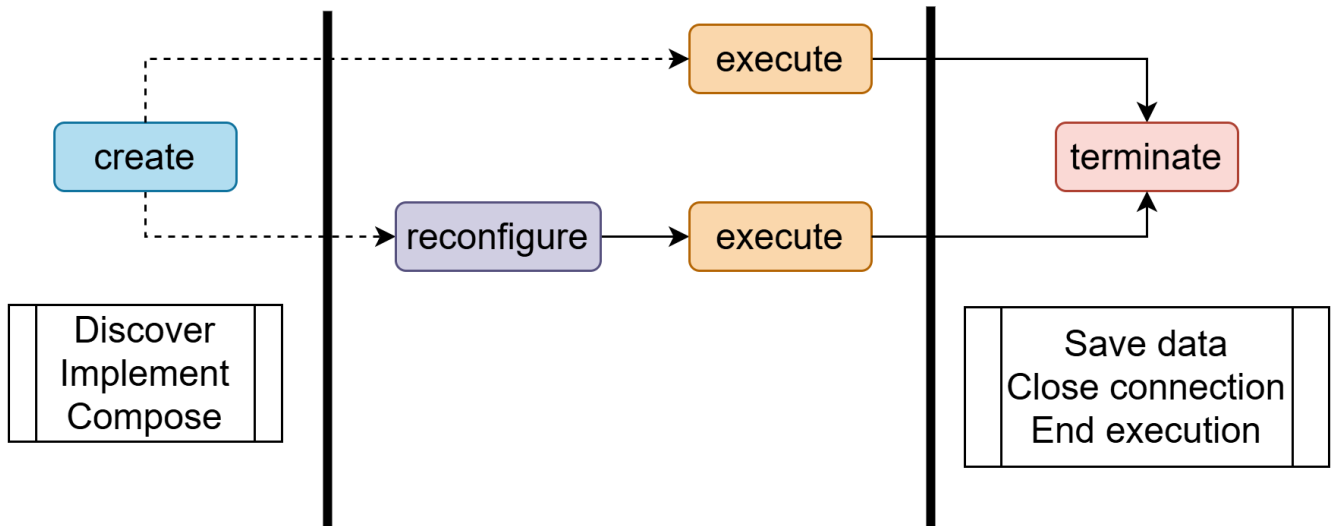
This feature requires specific installation setup.

1. Integrated GitLab installation

2. A valid GitLab repository for the logged in user. See an [example repository](#). This repository can be cloned and customised as needed.
3. A [linked GitLab Runner](#) to the user gitlab repository.

DT LIFECYCLE

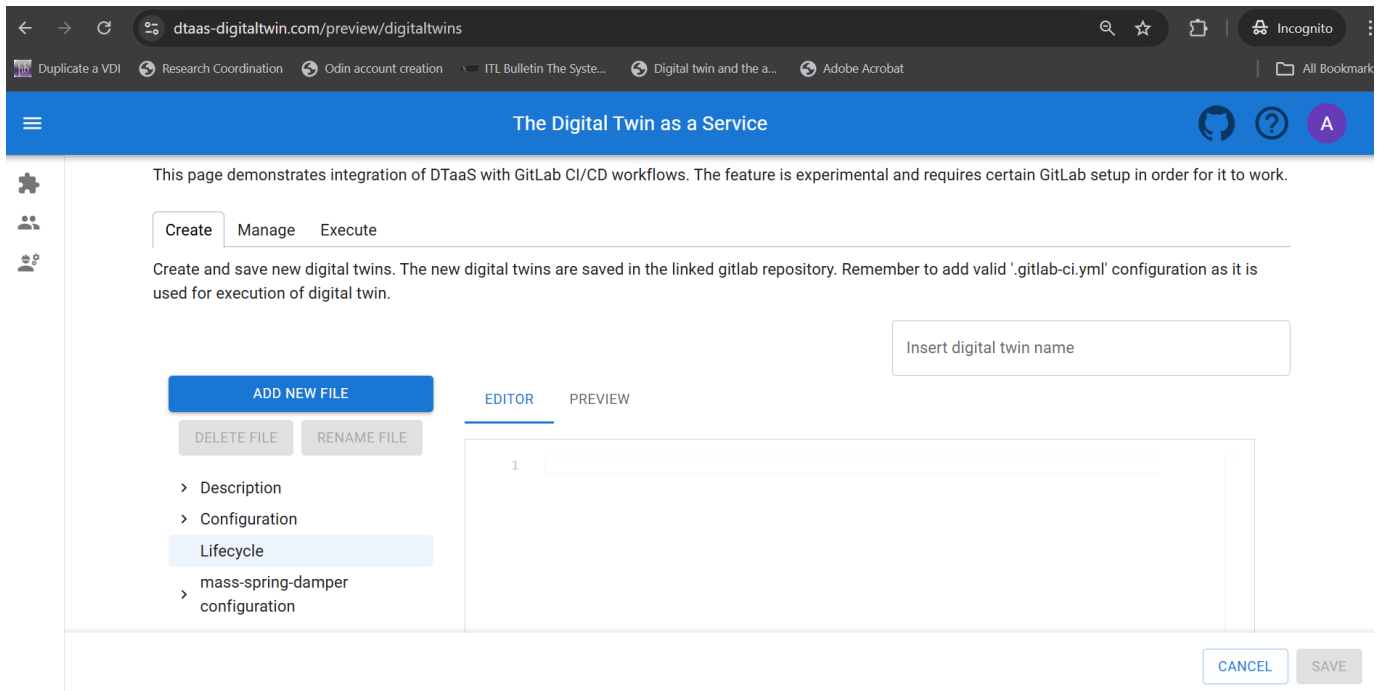
The DT preview implements the **Create**, **Manage** and **Execute** stages of a [DT lifecycle](#). The suggested sequence of use for different lifecycle stages are:



There are dedicated tabs for **Create**, **Manage** and **Execute** stages. The selection of DT assets for Create stage happens via [Library preview page](#). The Manage tab fulfills reconfigure feature. The Execute and Terminate are managed on the Execute tab.

CREATE TAB

The users select reusable DT assets and arrive on the Create tab. The following figure shows DT creation page after selecting *mass-spring-damper* as a reusable asset for the new DT.



The left-side menu shows the possibility of creating **necessary structure** and elements for a new DT. Each reusable asset selected for this new DT appears on the left menu. Its configuration can be updated as well.

These files on the left menu correspond to three categories.

- **Description:** contains *README.md* providing comprehensive description of DT and *description.md* providing a brief description. The brief description is shown in the DT tabs and clicking on the *Details* button shows the complete README.md. The *Details* button is available only on the Manage page.
- **Configuration:** Contains a *.gitlab-ci.yml* for running the required lifecycle scripts and operations of the DT. Additional json and yaml files can be added to create configuration for a new DT.
- **Lifecycle:** These are the DT lifecycle scripts.

The *Add New File* button can be used to add new files in all the three categories. Finally, click on *SAVE* button to save the new DT. Newly created DTs become immediately available on the **Manage** and **Execute** tabs.

MANAGE TAB

The screenshot shows the 'Manage' tab of the 'The Digital Twin as a Service' interface. The page title is 'The Digital Twin as a Service'. Below the title, there are tabs for 'Create', 'Manage', and 'Execute'. The 'Manage' tab is active. The main content area displays a list of digital twins. Each entry includes a title, a brief description, and three buttons: 'DETAILS', 'RECONFIGURE', and 'DELETE'. The digital twins listed are 'Hello world', 'Mass spring damper', 'Runner', 'Shm devops', and 'Yafem demo'. The 'Mass spring damper' entry is highlighted, showing its description: 'The mass spring damper digital twin (DT) comprises two mass spring dampers and demonstrates how a co-simulation based DT can be used'.

The manage tab allows for different operations on a digital twin:

- Checking the details (**Details** button)
- Delete (**Delete** button)
- Modify / Reconfigure (**Reconfigure** button)

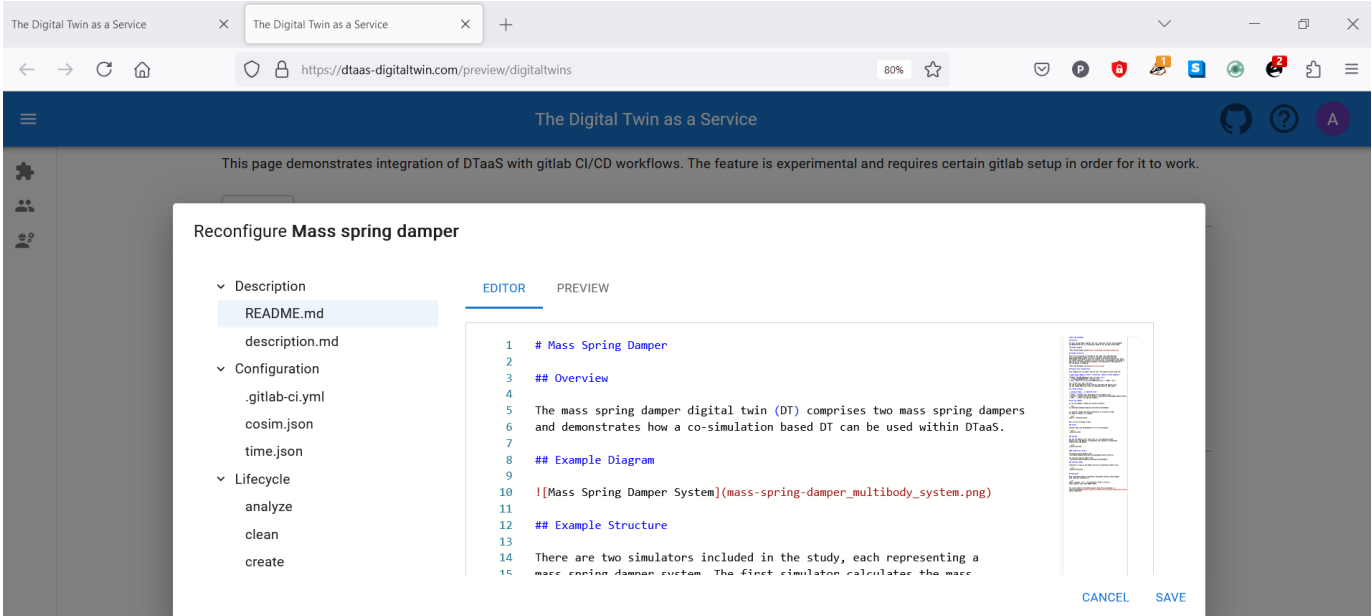
A digital twin placed in the DTaaS has a certain recommended structure. See the [assets](#) page for an explanation and [this example](#).

The information page shown using the Details button, shows the README.md information stored inside the digital twin directory.

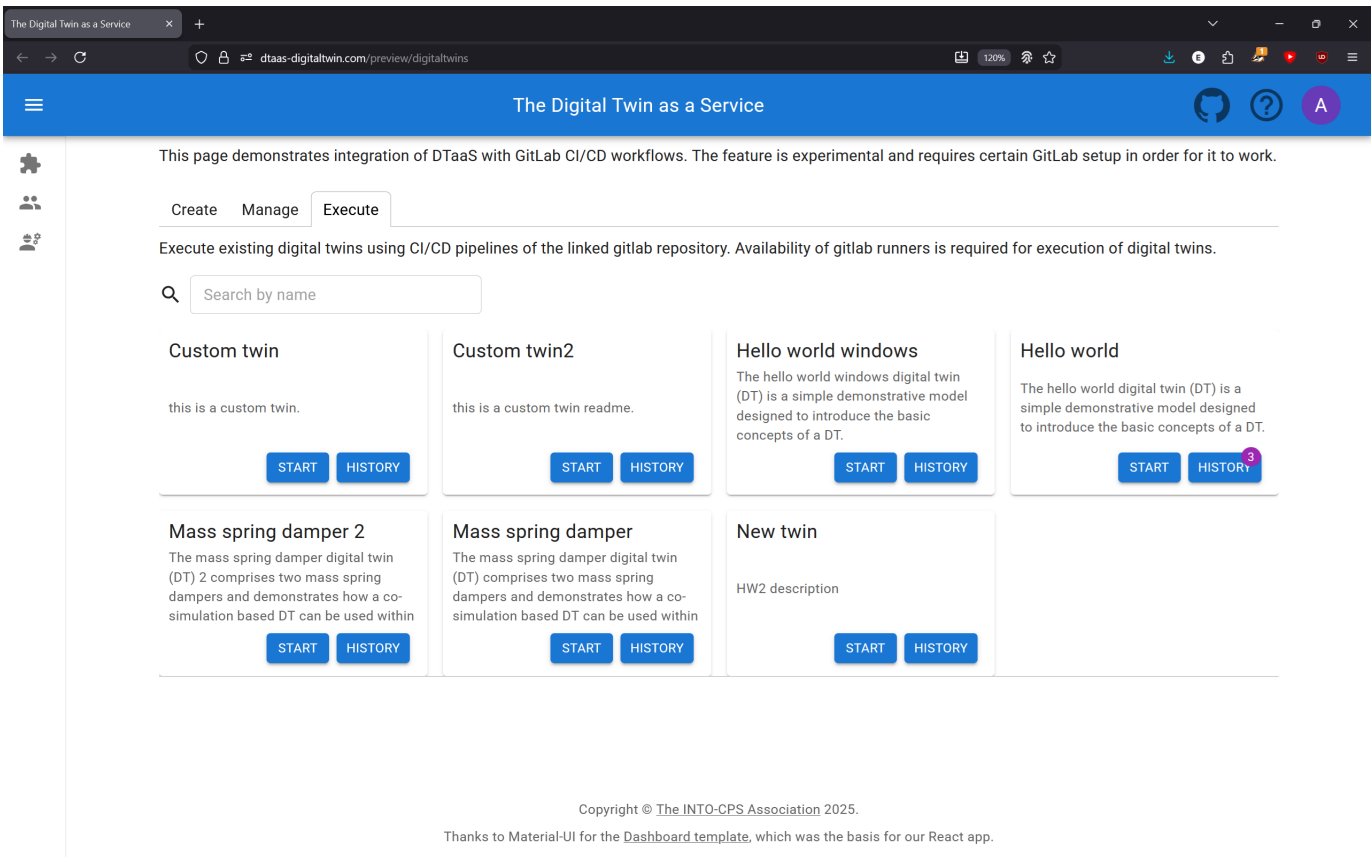
The screenshot shows the 'Details' page for the 'Mass Spring Damper' digital twin. The page has a title 'Mass Spring Damper' and an 'Overview' section. The overview text states: 'The mass spring damper digital twin (DT) comprises two mass spring dampers and demonstrates how a co-simulation based DT can be used within DTaaS.' Below the overview is an 'Example Diagram' section. The diagram shows two masses, m_1 and m_2 , connected by a spring with stiffness c_c and a damper with coefficient d_c . Mass m_1 is also connected to a fixed wall by a spring with stiffness c_1 and a damper with coefficient d_1 . Mass m_2 is connected to another fixed wall by a spring with stiffness c_2 . A force F_k is applied to mass m_1 . The displacement of mass m_1 is x_1 and the displacement of mass m_2 is x_2 . The diagram is enclosed in a dashed circle.

A reconfigure button opens an editor and shows all the files corresponding to a digital twin. All of these files can be updated. These files correspond to three categories.

- **Description**
- **Configuration**
- **Lifecycle**



EXECUTE TAB



The execute tabs shows the possibility of executing multiple digital twins. Once an execution of digital twin is complete, the execution log is also available.

Yafem demo log

execute_yafem

```
Running with gitlab-runner 17.5.3 (12030cf4)
on dtaas-runner-overture t1_zagU2y, system ID: r_GcJT1uRR1WRe
Preparing the "docker" executor
Using Docker executor with image ubuntu:24.04 ...
Pulling docker image ubuntu:24.04 ...
Using docker image sha256:bf16bdcff9c96b76a6d417bd8f0a3abe0e55c0ed9bdb3549e906834e2592fd5f for ubuntu:24.04 with
digest ubuntu@sha256:b59d21599a2b151e23eea5f6602f4af4d7d31c4e236d22bf0b62b86d2e386b8f ...
Preparing environment
Running on runner-t1zagu2y-project-2-concurrent-0 via 9e90c5018a4e...
Getting source from Git repository
Fetching changes with git depth set to 20...
Reinitialized existing Git repository in /builds/gitlab/dtaas/TestUserDTaaS/.git/
Checking out a8241698 as detached HEAD (ref is main)...
Removing digital_twins/yafem-demo/.venv/
Removing digital_twins/yafem-demo/requirements.txt
Removing digital_twins/yafem-demo/yafem-0.2.5-py3-none-any.whl
```

[CLOSE](#)

Setting Allowed Values

Some Setting values will cause problems if used. This document provides an in-depth examination of the values that are allowed and values that are not. Expected errors and troubleshooting steps for this page are also covered.

To understand what each parameter does, refer to the [settings](#) document.

Note: The user must have the appropriate access rights to all the resources connected to the application.

RUNNER TAG

- [Allowed values](#)

This parameter has to be a text string like "linux", "ubuntu", "2" and match some Runner's tags on the execution service:

Assigned project runners



Multiple values, like "linux, windows" is not supported and will be treated as one tag.

- [Not allowed values](#)

If the **Runner Tag** field *does not* match a Runner, no jobs will be picked up and the job will hence time out. If the twins unexpectedly time out, verify that the tag has been spelt correctly.

No tag (i.e. a blank field) is permitted by some services like GitLab, meaning that *any* Runner that is configured to pick up tag-less jobs can pick these jobs up. A limitation of DTaaS is that *some* tag is required. No tags, like other unallowed tags, will be ignored.

- [Visual examples](#)

Runner Tag

linux

Tag used for GitLab CI runners (e.g., linux, windows)

Runner Tag

linux, windows

Tag used for GitLab CI runners (e.g., linux, windows)

Runner Tag

Tag used for GitLab CI runners (e.g., linux, windows)

BRANCH

- [Allowed values](#)

The **Branch** field must be a text string matching a branch in the user *and* common library repositories. For example: "main" or "master":

dtaas / common

C

common



main

common /

+

Find file

Edit

Code

- [Not allowed values](#)

This field should not be left empty or set to a value not matching a branch in both repositories. While technically allowed, this may cause errors or unexpected behaviours.

Expected error:

An error occurred while fetching assets: GitbeakerRequestError: 404 Tree Not Found

- [Visual examples](#)

Branch Name

main

Default branch name for GitLab projects

Branch Name

foo

Default branch name for GitLab projects

Branch Name

Default branch name for GitLab projects

GROUP NAME

• Allowed values

The **Group Name** field again requires a text string, and it must further match some group in the storage service (e.g. GitLab).

Example:

DTaaS

Explore / Groups

Explore groups

[New group](#)

Below you will find all the groups that are public or internal. Contribute by requesting to join a group. [Learn more.](#)

🕒

Search or filter results...

🔍

Created date

↓

🏠

D

dtaas

🌐

👤 0

📁 26

🔗 2

Group names are case insensitive on GitLab, but matching the case exactly is recommended for consistency and to remove it as a source of error.

• Not allowed values

If the group does not exist or the field is left blank, errors will occur upon accessing Digital Twins, etc. Ensure that a value is entered.

Expected error:

An error occurred while fetching assets: GitbeakerRequestError: 404 Group Not Found

• Visual examples

Group Name

DTaaS

The group name used for GitLab operations

Group Name

foo

The group name used for GitLab operations

Group Name

The group name used for GitLab operations

COMMON LIBRARY PROJECT NAME

- Allowed values The **Common Library Project name** parameter is a text string. It must correlate to the repository responsible for keeping shared resources. Examples: "common" and "library":

dtaas / common



main

common /

+

Find file

Edit

Code

Failure to match a repository will break the DevOps page.

Expected error:

An error occurred while fetching assets: Error: Common project not found

- Not allowed values

This field should not be left blank or set to a value not matching a repository. It is inadvisable for it to match a user repository.

- Visual examples

Common Library Project name

common

Project name for the common library

Common Library Project name

foo

Project name for the common library

Common Library Project name

Project name for the common library

DT DIRECTORY

- Allowed values

Like the other fields, this must be a text string. It should match the name of the folder within the common library and most importantly the user repository where the Digital Twins are stored.

C

common

main

common /

+

Find file

Edit

Code



Merge branch 'name-change' into 'main'

...

299d6e62



History

Prasad Talasila authored 2 years ago

Name	Last commit	Last update
 data	Add new directory	2 years ago
 digital_twins	rename directory	2 years ago
 functions	Update file function1.txt	2 years ago
 models	Add new directory	2 years ago
 tools	Add new directory	2 years ago
 README.md	Initial commit	2 years ago

- Not allowed values

Do not leave this blank or erroneously mapped. This will lead to silent failure with the Digital Twins not loading or Digital Twins from a previous set up to show up.

DT Directory

digital_twins

Directory for Digital Twin files

DT Directory

foo

Directory for Digital Twin files

DT Directory

Directory for Digital Twin files

 TROUBLESHOOTING

The Digital Twins are not showing up:

Verify that Group, Branch, common and DT directory all are configured exactly, like it is on the backend.

The Digital Twins are timing out:

Verify that the Runner Tag parameter is not a list of tags, that the runner is active and matches its tag.

 SUMMARY

The possible values for correctly connecting DTaaS to the relevant storage and service instances have been specified, along with potential pitfalls. The key point is not to leave any of the fields blank; they should be of the text string format and properly correspond to the relevant services and runners. Some troubleshooting steps have further been provided to overcome common hurdles.

Capabilities

In addition to creating and editing Digital Twins, it is also possible to run them and adjust their execution parameters. The DTaaS permits running multiple Digital Twins simultaneously and even changing settings while they are running, without the need to manually fetch the data: it will load upon returning to whichever branch and group was used for execution. The settings include both repository and execution options, while all twins can readily be run and checked from the Execution tab under the Preview page.

Read more about [Settings](#), [Setting Values](#) and [Concurrent Execution](#) on their respective pages.

The following sections describe the specific capabilities of each of these features.

 SETTINGS

The table below describes the different test setups (valid, invalid, etc.), the resultant behaviour and what was deemed to be expected behaviour across the features. Following every table, there is a summary and list of potential problems.

Setting	Expected behaviour	Observed behaviour	Test method
Runner tag valid	Job is picked up by runner with relevant tag and completes without error.	✔ Same as expected.	Goto Preview page. Go to Account. Change Runner Tag. Go back one page. Execute Hello world twin. Verify runner name based on tag (repeat with 2nd runner)
Runner tag invalid	Job is never picked up. Runner times out after 10 minutes.	✔ Same as expected.	Change Runner Tag to "foo" (inexistent). Run Hello World twin.
Runner Tag no value	Configuration saves. Running a twin succeeds if appropriate runner exists.	✘ Not picked up, times out.	Set <code>run_untagged = false</code> in local gitlab runner config. Mark "Run untagged jobs" as true in gitlab instance (https://dtaas-digitaltwin.com/gitlab). Change Runner Tag to the empty string on application. Run Hello world twin.
Branch valid	Job runs with the correct branch and completes without error.	✔ Same as expected.	Make new branch in GitLab instance project. Change Branch name to "master-2". Execute Hello world twin. Verify ref matches branch in execution log.
Branch invalid	Execution tab gracefully displays no twins as branch does not exist.	✘ IF twins are not cached: Throws an error displayed to user: An error occurred while fetching assets: GitbeakerRequestError. IF twins are cached: Job fails. Snackbar says "Execution error for [twin name]". No log available in execution history.	Change Branch name to "master-1" (inexistent). IF twins are cached: Execute Hello world twin.
Group name valid	The twin is runnable.	✔ Same as expected.	Click "reset to defaults". Run twin.
Group name invalid	Execution tab gracefully displays no twins as group does not exist.	✘ Same as Branch invalid case.	Change group name to "Foo" (inexistent). IF twins are cached: Execute Hello world twin.
Common name valid	Library twins are visible.	✔ Same as expected. *Private twins also show up in Common twins.	Click "reset to defaults". Goto library. Goto common twins. Inspect twin visibility.
Common name invalid	Library twins gracefully not visible.	✘ Displays error: An error occurred while fetching assets: Error: Common project foo not found	Change Common Library Project name to "Foo" (inexistent). Goto common twins. Inspect visibility.
DT directory valid	Twins are visible.	✔ Same as expected.	Click "reset to defaults". Goto library. Goto Execution tab. Inspect twin visibility.
DT directory name invalid	Twins gracefully not visible.	✘ Depends on the cache. Displays error if there is none: An error	Change Common DT directory name to "Foo" (inexistent). Goto

Setting	Expected behaviour	Observed behaviour	Test method
		occurred while fetching assets: GitbeakerRequestError	Execution tab. Inspect twin visibility.
Group Name, DT Directory, Common Library Project Name, Branch name no value	Invalid value caught early. Cannot save, appropriate feedback of invalid form fill out.	❌ Same as Branch invalid case.	Change Group Name, DT Directory, Common Library Project Name, Branch name to the empty string.

Summary: Changing to valid settings works. Invalid settings in the best case display an error in the HTML (no cache) and otherwise shows stale twins.

Problems:

- Stale state maintained after settings are updated unless page is refreshed.
- Displaying technical errors to the user when the settings are invalid.
- Permitting invalid values such as empty strings when they are required
- No tags runners may not work, but possibly local problem.
- Private twins show up as common twins

CONCURRENT EXECUTION

Expected behaviour	Observed behaviour	Test method
Both twins run successfully simultaneously.	✅ Same as expected.	Run the same twin twice at the same time.
All twins run successfully simultaneously.	✅ Same as expected.	Run different twins at the same time.

Concurrent Execution Across Different Runners

Expected behaviour	Observed behaviour	Test method
Both twins run successfully simultaneously. They report the correct runner name in the Execution logs.	✅ Same as expected.	Set up 2 ubuntu GitLab runners with different tags. Change Runner Tag to first runner's tag. Execute twin. Change Runner Tag to second runner tag. Execute another twin.
History stays after editing, it is still executing.	✅ Same as expected.	Execute → Edit settings → Execute. Check - is history still there? Does it look correct?

Summary:

All functionality works according to the tests. The logs are still there after changing settings and running any combination of twins and runners work (Only Ubuntu runners tested).

IMPLEMENTING BACKENDS

The DTaaS is by default set up to work with GitLab as execution and storage backend, but other combinations may be more suitable. As such, the code base is designed with this flexibility in mind, so reimplementing from scratch is not required when a new backend is needed. In future versions of the DTaaS, more backends may be available by default, such as GitHub and Azure.

SUMMARY

Digital Twins can be queued as needed and live settings changes will not interfere with data retention, while enabling the testing of multiple setups simultaneously. For greater flexibility, those with the requisite technical expertise can readily expand upon the available backends.

Running Multiple Digital Twins at the Same Time

The DTaaS platform allows for executing multiple Digital Twins in tandem with one another. This can save hours of time when dealing with intensive Digital Twins or when deploying them across different resources.

RUNNING DIGITAL TWINS

To deploy a Digital Twin, navigate to the *Execution tab* on the *Digital Twin Preview page*. This can be found by clicking on the

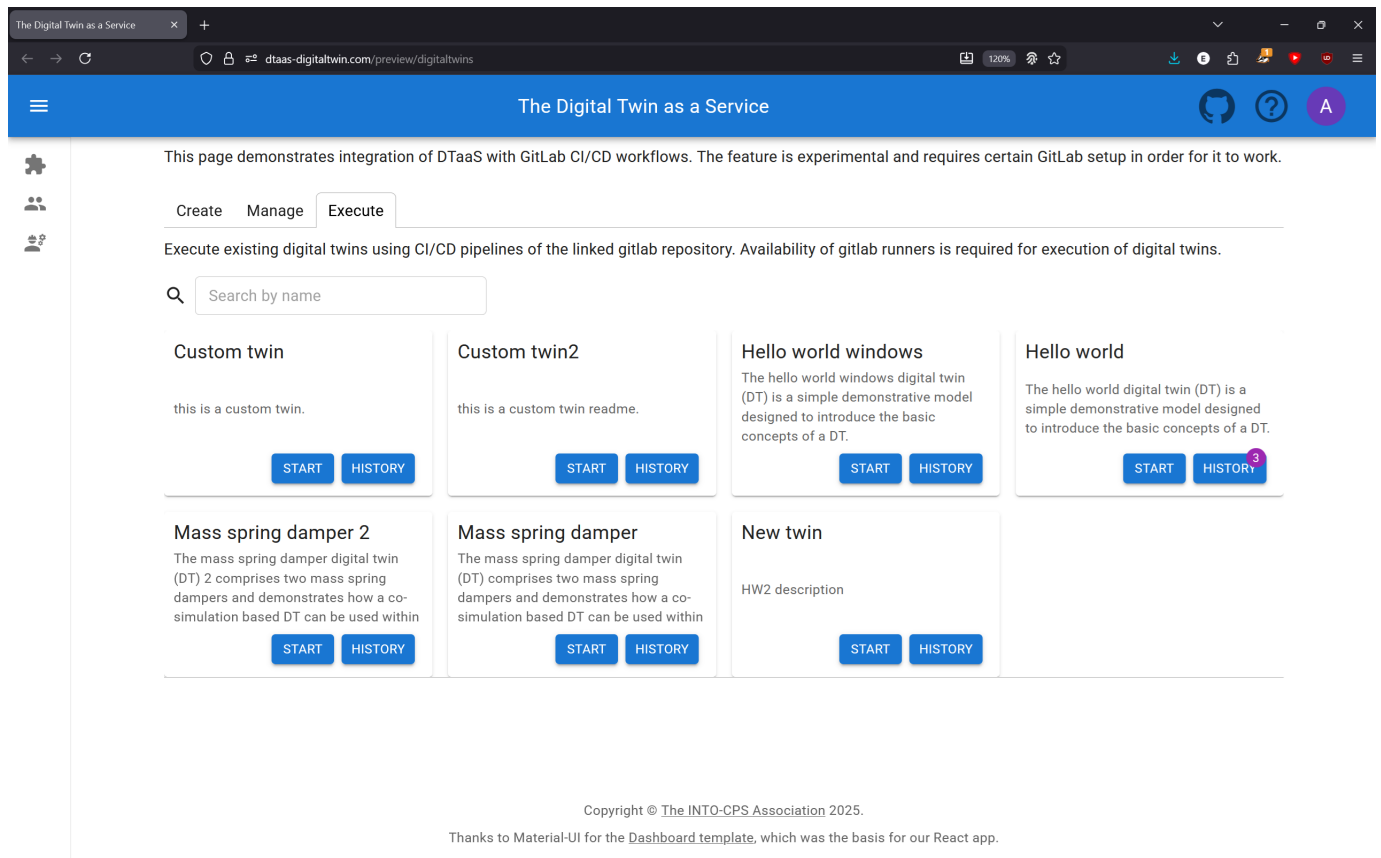


Workbench icon in the sidebar on the left hand side and locating *Digital Twins Preview* among the other links. Click to open the *Digital Twin Preview page* in a new tab.

In the newly opened tab, a series of cards is displayed with a name, short description of the Digital Twins and **START** and **HISTORY** buttons. The **START** button sends a *job* to the backend, telling it to run the simulation associated with that Digital Twin, which is defined by its [Life Cycle files](#). These can be inspected and changed in the *Edit* tab.

Running Multiple Twins

Pressing the **START** button multiple times queues multiple simulations. It is also possible to queue different kinds of Digital Twins.



The deployment service (e.g. GitLab) automatically load balances across available runners, so it can be configured once and left to run autonomously.

THE EXECUTION LOG

Pushing the **HISTORY** button provides an overview of all current and past simulation trials, distinguished by time of execution and status (Running, Failed, Succeeded or Timed Out). To review the results of a simulation, click on an entry (or the arrow on the right of the entry) to expand it. Each step in the Life Cycle will then be presented.

To delete an entry, click the trash icon. A verification dialogue will confirm the entry to be deleted. All entries of a Digital Twin can also be deleted with the **CLEAR ALL** button.

Click the stop symbol in the log to stop an execution:

Hello world Execution History

Execution History

24/12/2025, 16.24.15

Status: Running

24/12/2025, 16.24.11

Status: Running

24/12/2025, 16.24.09

Status: Running

CLEAR ALL

CLOSE

 CHANGING RUNNERS

The runners that pick up jobs can be changed in the settings. Further information on [these and other settings](#) is available.

 SUMMARY

This document has described how to navigate the DevOps Execution page: executing multiple Digital Twins simultaneously and inspecting and deleting their logs.

- 72/138 -

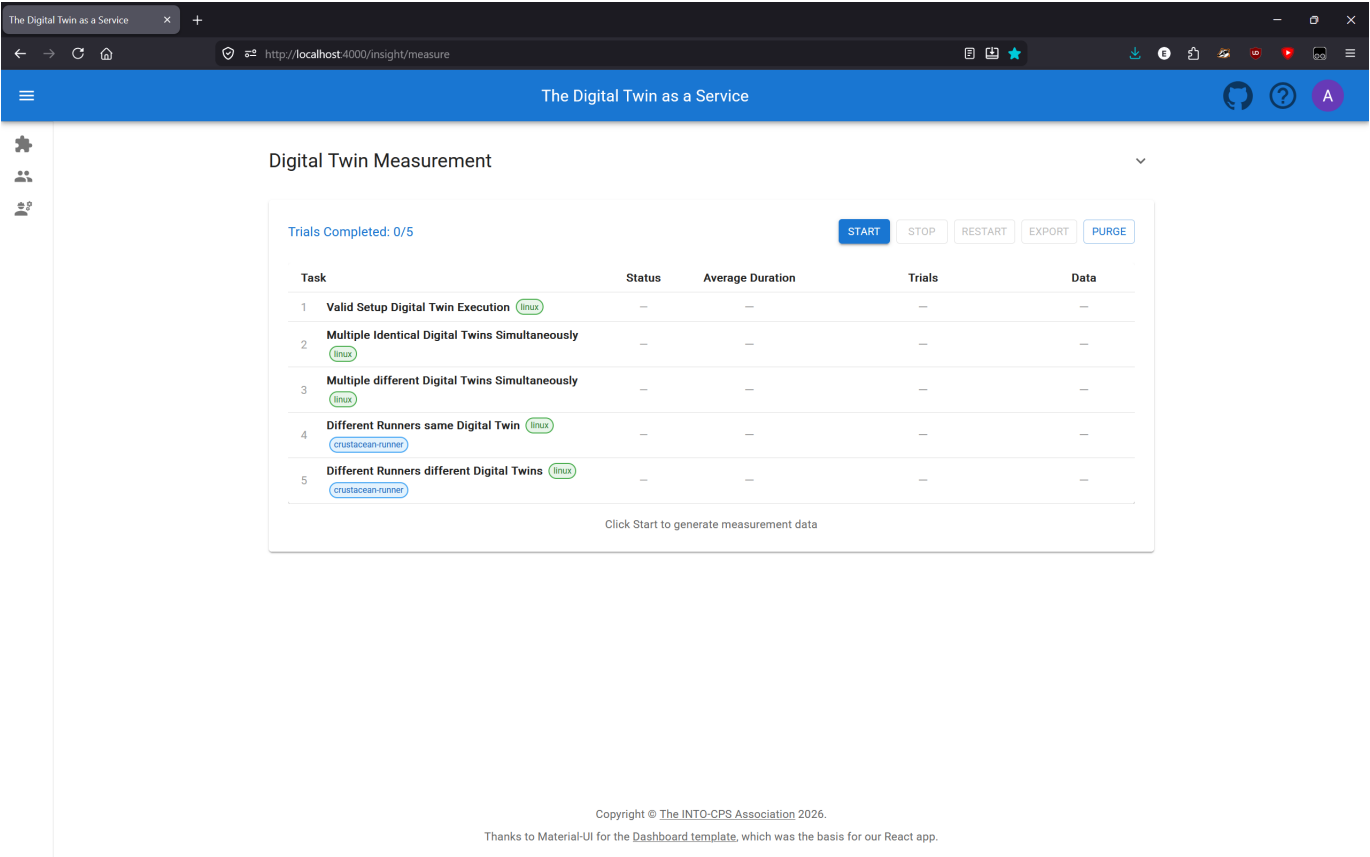
Copyright © 2022 - 2026 The INTO-CPS Association

Performance Measurement

The DTaaS provides a built-in performance measurement tool that allows developers to measure the execution time of Digital Twin features. This is useful for understanding performance characteristics and establishing baselines for future optimization.

ACCESSING THE MEASUREMENT PAGE

Navigate to `http://localhost:4000/insight/measure` or `https://intocps.org/insight/measure` to access the measurement page. This page is only accessible to authenticated users.



FEATURES

Measurement Tasks

The measurement suite includes the following tasks that measure different aspects of Digital Twin execution:

Task	Description
Valid Setup Digital Twin Execution	Runs the primary Digital Twin with current setup
Multiple Identical Digital Twins Simultaneously	Runs the primary Digital Twin twice at once
Multiple different Digital Twins Simultaneously	Runs the primary and secondary Digital Twins at once
Different Runners same Digital Twin	Runs the primary Digital Twin twice with 2 different runners
Different Runners different Digital Twins	Runs the primary and secondary Digital Twins with 2 different runners

By default, the primary Digital Twin is `hello-world` and the secondary is `mass-spring-damper`. Both can be changed in the measurement settings.

Digital Twin Measurement

Trials Completed: 1/5

START

STOP

RESTART

EXPORT

PURGE

Task	Status	Average Duration	Trials	Data
1 Valid Setup Digital Twin Execution linux	SUCCESS	26.9s	Trial 126.9s hello-world (Pipeline: 15243) success Runner: linux	Download Task Results
2 Multiple Identical Digital Twins Simultaneously linux	RUNNING	—	Trial 1— hello-world (Pipeline: 15245) Parent pipeline starting Runner: linux hello-world — Runner: linux	—
3 Multiple different Digital Twins Simultaneously linux	PENDING	—	—	—
Different Runners same Digital Twin linux	Measurement data generation in progress			

Configuration Options

Configuration is managed in the **Measurement Settings** section of the account settings page.

- **Trial Number:** Number of times each task is repeated to calculate average execution time (default: 3). Adjust this for more or fewer data points.
- **Measurement Secondary Runner Tag:** The runner tag used for multi-runner measurement tests. The primary runner tag is configured separately in the account settings.
- **Primary Digital Twin Name:** The Digital Twin used in single-DT and same-DT measurement tasks (default: `hello-world`).
- **Secondary Digital Twin Name:** The Digital Twin used as the second DT in multi-DT measurement tasks (default: `mass-spring-damper`).

Note: the Primary and Secondary Digital Twin dropdowns need to load the available Digital Twins the first time the settings page is visited in a session. They will appear disabled until loading completes.

The screenshot shows the 'Settings' page of 'The Digital Twin as a Service' application. The 'Measurement Settings' section is highlighted with a red box. It contains the following fields:

- Trial Number:** A numeric input field with the value '3'. Below it, a description reads: 'Number of times each task is repeated to calculate average execution time'.
- Measurement Secondary Runner Tag:** A text input field with the value 'windows'. Below it, a description reads: 'Runner tag used for multi-runner measurement tests'.
- Primary Digital Twin:** A dropdown menu with the selected value 'hello-world'. Below it, a description reads: 'Digital Twin used in single-DT and same-DT measurement tasks'.
- Secondary Digital Twin:** A dropdown menu with the selected value 'mass-spring-damper'. Below it, a description reads: 'Digital Twin used as the second DT in multi-DT measurement tasks'.

At the bottom of the 'Measurement Settings' section, there are two buttons: 'RESET TO DEFAULTS' and 'SAVE SETTINGS'.

Controls

- **Start:** Begin the measurement suite from the first task
- **Stop:** Cancel all running executions (shows confirmation dialog)
- **Restart:** Discard all current results and start fresh (shows confirmation dialog)
- **Export:** Download all measurement results as a JSON file (available when results exist)
- **Purge:** Clear all stored measurement data from the database

UNDERSTANDING RESULTS

Status Values

Status	Meaning
PENDING	Task has not started yet
RUNNING	Task is currently executing
SUCCESS	All trials completed successfully
FAILURE	One or more trials failed
STOPPED	Task was interrupted by user

Metrics

- **Average Duration:** Mean execution time across all trials for each task
- **Total Time:** Overall time for the complete measurement suite (shown after completion)

Measurement Table Columns

The measurement results table displays the following columns:

Column	Description
Task	Task number, name, and description
Status	Current execution status (NOT_STARTED, PENDING, RUNNING, SUCCESS, FAILURE, or STOPPED)
Average Duration	Mean execution time across all completed trials, displayed in seconds
Trials	Visual cards showing each trial's execution details and status. Each trial represents one iteration of the task
Data	Download button to export individual task results as JSON

Each row can be clicked to expand it and show a toggle alongside the task description. Disabling a task skips it during execution.

DATA STORAGE

Measurement data is stored separately from regular execution history in the browser's IndexedDB. This enables the following:

- Keep measurements separate from normal DT execution logs
- Purge measurement data without affecting execution history
- Download results as JSON for further analysis

DOWNLOADING RESULTS

Per-Task Download

After a task completes successfully, click "Download Task Results" in the Data column to export that specific task's measurements as JSON.

Full Results Download

Use the **Export** button in the controls toolbar to download all measurement results as JSON. This button is available whenever results exist.

Both options shown below, from a completed measurement run.

Digital Twin Measurement

Trials Completed: 5/5

START

STOP

RESTART

EXPORT

PURGE

Task	Status	Average Duration	Trials	Data
1 Valid Setup Digital Twin Execution linux	SUCCESS	27.0s	Trial 1 27.0s hello-world (Pipeline: 15313) success Runner: linux	Download Task Results
2 Multiple Identical Digital Twins Simultaneously linux	SUCCESS	51.3s	Trial 1 51.3s hello-world (Pipeline: 15315) success Runner: linux hello-world (Pipeline: 15317) success Runner: linux	Download Task Results
			Trial 1 632.9s hello-world (Pipeline: 15319) success	

Completed in 711.8s
Measurement data generation complete

JSON Format

The exported JSON contains detailed information about each task and its trials. Each trial represents one iteration of a task, and contains an `executions` array with the pipeline execution details. The configuration is shared at the task level, while each execution includes its runner tag.

The exported JSON follows the structure below:

```

1  {
2    "totalTimeSeconds": number,
3    "tasks": [
4      {
5        "Task Name": string,
6        "Description": string,
7        "config": {
8          "Branch name": string,
9          "Group name": string,
10         "Common Library project name": string,
11         "DT directory": string
12       },
13       "trials": [
14         {
15           "Time Start": "ISO8601 timestamp",
16           "Time End": "ISO8601 timestamp",
17           "Status": "PENDING | RUNNING | SUCCESS | FAILURE | STOPPED",
18           "executions": [
19             {
20               "dtName": string,
21               "pipelineId": number,
22               "status": string,
23               "Runner tag": string
24             }
25           ]
26         }
27       ],
28       "Average Time (s)": number,
29       "Status": string
30     }
31   ]
32 }
```

TIMING BEHAVIOUR

The measurement system uses deliberate delays of 750ms to avoid overloading the GitLab instance with simultaneous requests:

- **Between trials:** A 750ms delay is applied before each trial *except the first*. Trial 0 starts immediately; trials 1, 2, ... each wait 750ms after the previous trial completes.
- **Between executions within a trial:** When a task runs multiple Digital Twins (e.g. "Multiple different Digital Twins Simultaneously"), each execution is staggered by 750ms. The first execution starts immediately, the second after 750ms, the third after 1500ms, and so on.

When interpreting the **Average Duration** of a task, keep in mind that the stagger delay between executions is included in the measured time. The delay between trials is *not* included, since each trial's timer starts after its delay.

BEST PRACTICES

1. **Run measurements during low-usage periods** to get consistent results
2. **Use multiple iterations** (3-5) for more reliable averages
3. **Ensure runners are available** before starting the measurement
4. **Navigating away** from the measurement page while it is running is permitted — execution continues in the background. However, **changing the URL, refreshing, or closing the tab will stop the measurement**. Notifications indicate when the measurement completes or is stopped.

TROUBLESHOOTING

Issue	Solution
Task appears stuck, old pipelines appears	Reauthenticate the app by refreshing the tab
Tasks time out	Verify runners are online, uses the tag and are accessible

RELATED DOCUMENTATION

- [Concurrent Execution](#)
- [Execution Settings](#)
- [Capabilities](#)

3.6 Working with GitLab

The DTaaS platform relies on GitLab for two purposes:

- 1. OAuth 2.0 authorization service
- 2. DevOps service

The [admin](#) documentation covers the OAuth 2.0 authorization configuration. This guide addresses the use of git commands and project structure for the GitLab DevOps service within the DTaaS.

3.6.1 Preparation

The first step is to create a GitLab project with the *username* in the GitLab user group named *dtaas*.



dtaas / New project / Create blank project



Create blank project

Create a blank project to store your files, plan your work, and collaborate on code, among other things.

Project name

user1

Must start with a lowercase or uppercase letter, digit, emoji, or underscore. Can also contain dots, pluses, dashes, or spaces.

Project URL

https://shared.dtaas-digitaltwin.com/gitlab/

dtaas

Project slug

user1

Want to organize several dependent projects under the same namespace? [Create a group](#).

Visibility Level

☒ Private

Project access must be granted explicitly to each user. If this project is part of a group, access is granted to members of the group.

Project Configuration

☐ Initialize repository with a README

Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

☐ Enable Static Application Security Testing (SAST)

Analyze your source code for known security vulnerabilities. [Learn more](#).

Create project

Cancel

The user must have ownership permissions over the project.

dtaas > user1 > **Members**

Project members

Import from a project

Invite a group

Invite members

You can invite a new member to **user1** or invite another group.

Members 2

Filter members

Account

Account

Filter members

Account	Source	Max role	Expiration	Activity
 root @root	dtaas	Owner	Expiration date	User created: Dec 18, 2024 Access granted: Dec 18, 2024 Last activity: Dec 18, 2024
 user1 @user1	Direct member by Administrator	Owner	Expiration date	User created: Dec 19, 2024 Access granted: Dec 19, 2024



Warning

The DTaaS website expects a default branch named `main` to exist. The website client performs all git operations on this branch. The preferred default branch name can be changed in the [settings page](#).

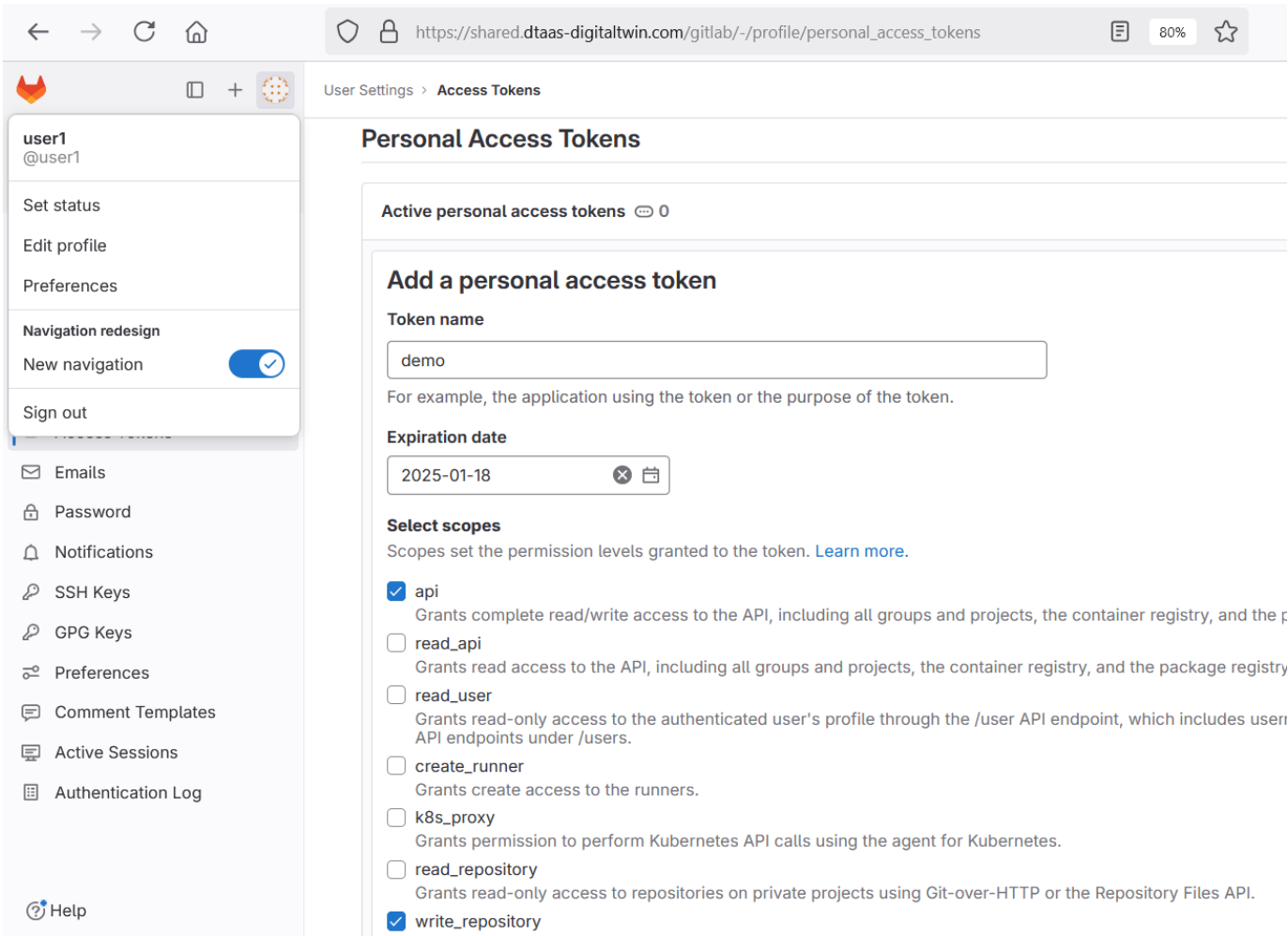
3.6.2 Git commands

Standard git commands and workflows should be utilised. There are two methods for using a GitLab project as a remote git server:

- Over SSH using a personal SSH key
- Over HTTPS using [personal access tokens \(PAT\)](#)

This tutorial demonstrates the use of PAT for working with the GitLab server.

The first step is to create a PAT.



This token should be copied and used to clone the git repository.

3.6.3 Library Assets

GitLab is used to store the reusable **Library** assets of all users. A [mandatory structure](#) exists for storing and using Library assets including digital twins. A properly initialized GitLab project should have the following structure.

dtaas > user1



user1
Project ID: 2 [Leave project](#)

🔔

☆ Star 0

🍴 Forks 0

🔗 1 Commit 🌿 1 Branch 🏷️ 0 Tags 💾 2 KiB Project Storage



adds sample project structure
prasadtalasila authored just now

d679b664 

main ▾ user1 / + ▾ History Find file Edit ▾ 📄 ▾ **Clone ▾**

 README  CI/CD configuration  Add LICENSE  Add CHANGELOG  Add CONTRIBUTING

Auto DevOps enabled  Add Kubernetes cluster  Add Wiki  Configure Integrations

Name	Last commit	Last update
📁 data	adds sample project structure	5 minutes ago
📁 digital_twins	adds sample project structure	5 minutes ago
📁 functions	adds sample project structure	5 minutes ago
📁 models	adds sample project structure	5 minutes ago
📁 tools	adds sample project structure	5 minutes ago
🔥 .gitlab-ci.yml	adds sample project structure	just now
📄 README.md	adds sample project structure	5 minutes ago

Pay special attention to `.gitlab-ci.yml`. It must be a valid GitLab DevOps configuration. The [example repo](#) provides a sample structure.

For example, with `PAT1` as the PAT for the `dtaas/user1` repository, the command to clone the repository is:

```
1 $git clone \
2   https://user1:PAT1@dtaas-digitaltwin.com/gitlab/dtaas/user1.git
3 $cd user1
```

After adding the required Library assets:

```
1 $git push origin
```

3.6.4 Next Steps

A [GitLab runner](#) should be integrated with the project repository. Runners may already be installed with the DTaaS platform. These can be verified on the runners page. Additionally, [custom runners](#) can be installed and integrated with the repository.

The [Digital Twins Preview](#) can then be used to access the DevOps features of the DTaaS platform.

3.7 Runner

A utility service to manage safe execution of remote scripts / commands. The user launches this from the command line and lets the utility manage the commands to be executed.

The runner utility runs as a service and provides REST API interface to safely execute remote commands. Multiple runners can be active simultaneously on one computer. The commands are sent via the REST API and are executed on the computer with active runner.

Warning

This npm package works only on Linux platforms

3.7.1 Install

NPM Registry

The package is available on [npmjs](https://www.npmjs.com/package/@into-cps-association/runner).

Install the package with the following command:

```
1 sudo npm install -g @into-cps-association/runner
```

GitHub Registry

The package is available in GitHub [packages registry](https://github.com/marketplace/packages).

Set the registry and install the package with the following commands

```
1 sudo npm config set @into-cps-association:registry \
2   https://npm.pkg.github.com
3 sudo npm install -g @into-cps-association/runner
```

The `npm install` command asks for username and password. The username is the GitHub username and the password is the GitHub [personal access token](#). In order for the npm to download the package, the personal access token needs to have `read:packages` scope.

3.7.2 Configure

The utility requires config specified in YAML format. The template configuration file is:

```
1 port: 5000
2 location: 'script' #directory location of scripts
3 commands: #list of permitted scripts
4   - create
5   - execute
6   - terminate
```

It is suggested that the configuration file be named as `runner.yaml` and placed in the directory in which the `runner` microservice is run.

The `location` refers to the relative location of the scripts directory with respect to the location of `runner.yaml` file.

However, there is no limitation on either the configuration filename or the `location`. The path to `runner.yaml` can either be relative or absolute path. However, the `location` path is always relative path with respect to the path of `runner.yaml` file.

 The commands must be executable. Ensure that the commands have execute permission on Linux platforms.

3.7.3 Create Commands

The runner requires commands / scripts to be run. These need to be placed in the `location` specified in `runner.yaml` file. The location must be relative to the directory in which the **runner** microservice is being run.

3.7.4 Use

Display help.

```

1 $runner -h
2 Usage: runner [options]
3
4 Remote code execution for humans
5
6 Options:
7   -v --version           package version
8   -c --config <string>  runner config file specified in yaml format (default: "runner.yaml")
9   -h --help             display help

```

The config option is not mandatory. If it is not used, **runner** looks for `runner.yaml` in the directory from which it is being run. Once launched, the utility runs at the port specified in `runner.yaml` file.

```

1 runner #use runner.yaml of the present working directory
2 runner -c FILE-PATH #absolute or relative path to config file
3 runner --config FILE-PATH #absolute or relative path to config file

```

If launched on one computer, the service is accessible at `http://localhost:<port>`.

Access to the service on the network is available at `http://<ip or hostname>:<port>/`.

Application Programming Interface (API)

Three REST API methods are active. The route paths and the responses given for these two sources are:

REST API Route	HTTP Method	Return Value	Comment
localhost:port	POST	Returns the execution status of command	Executes the command provided. Each invocation appends to <i>array</i> of commands executed so far.
localhost:port	GET	Returns the execution status of the last command sent via POST request.	
localhost:port/history	GET	Returns the array of POST requests received so far.	

POST REQUEST TO /

Executes a command. The command name given here must exist in *location* directory.

Valid HTTP Request	HTTP Response - Valid Command	HTTP Response - Inalid Command
<pre>1 POST / HTTP/1.1 2 Host: intocps.org 3 Content-Type: application/json 4 Content-Length: 388 5 6 { 7 "name": "<command-name>" 8 }</pre>	<pre>1 Connection: close 2 Content-Length: 134 3 Content-Type: application/json; charset=utf-8 4 Date: Tue, 09 Apr 2024 08:51:11 GMT 5 Etag: W/"86-ja15r8P5HJu72JcR0fBTv4sAn2I" 6 X-Powered-By: Express 7 8 { 9 "status": "success" 10 }</pre>	<pre>1 Connection: close 2 Content-Length: 28 3 Content-Type: application/json; charset=utf-8 4 Date: Tue, 09 Apr 2024 08:51:11 GMT 5 Etag: W/"86-ja15r8P5HJu72JcR0fBTv4sAn2I" 6 X-Powered-By: Express 7 8 { 9 "status": "invalid command" 10 }</pre>

GET REQUEST TO /

Shows the status of the command last executed.

Valid HTTP Request	HTTP Response - Valid Command	HTTP Response - Invalid Command
<pre>1 GET / HTTP/1.1 2 Host: intocps.org 3 Content-Type: application/json 4 Content-Length: 388 5 6 { 7 "name": "<command-name>" 8 }</pre>	<pre>1 Connection: close 2 Content-Length: 134 3 Content-Type: application/json; charset=utf-8 4 Date: Tue, 09 Apr 2024 08:51:11 GMT 5 Etag: W/"86-ja15r8P5HJu72JcR0fBTv4sAn2I" 6 X-Powered-By: Express 7 8 { 9 "name": "<command-name>", 10 "status": "valid", 11 "logs": { 12 "stdout": "<output log of command>", 13 "stderr": "<error log of command>" 14 } 15 }</pre>	<pre>1 Connection: close 2 Content-Length: 70 3 Content-Type: application/json; charset=utf-8 4 Date: Tue, 09 Apr 2024 08:51:11 GMT 5 Etag: W/"86-ja15r8P5HJu72JcR0fBTv4sAn2I" 6 X-Powered-By: Express 7 8 { 9 "name": "<command-name>", 10 "status": "invalid", 11 "logs": { 12 "stdout": "", 13 "stderr": "" 14 } 15 }</pre>

GET REQUEST TO /HISTORY

Returns the array of POST requests received so far. Both valid and invalid commands are recorded in the history.

<pre>1 [2 { 3 "name": "valid command" 4 }, 5 { 6 "name": "valid command" 7 }, 8 { 9 "name": "invalid command" 10 } 11]</pre>

3.8 Examples

3.8.1 DTaaS Examples

Several example digital twins have been created for the DTaaS platform. These examples can be explored, and the steps provided in this **Examples** section can be followed to experience features of the DTaaS platform and understand best practices for managing digital twins within the platform. The following [slides](#) and [video](#) provide an overview of these examples.

Please see the following demos illustrating the use the DTaaS in two projects:

Project	Slides and Videos
CP-SENS project	Project Introduction: slides and video
	Wind turbine testing with the demo inside user workspace
	Python package and demo video
	Videos demonstrating digital twin for structural health monitoring applications: Data Acquisition System: Part-1 , Data Acquisition System: Part-2 , Operational Model Analysis , and Model Updating
Incubator	video

Workspace Examples

These examples run interactively inside the DTaaS user workspace. See the [workspace examples overview](#) for the full list and instructions on downloading and running them.

Automation Examples

These examples demonstrate the use of GitLab DevOps pipelines to automate the execution of digital twin lifecycle scripts within the DTaaS platform. See the [automation examples overview](#) for details.

↓ [DTaaS examples](#)

3.8.2 Workspace

Workspace Examples

These examples demonstrate digital twins that run interactively inside the DTaaS user workspace. Each example can be downloaded and executed using the lifecycle scripts provided.

COPY EXAMPLES

Copy all example code into the user workspace using the provided script:

```
1 wget https://raw.githubusercontent.com/INTO-CPS-Association/DTaaS-examples/main/getExamples.sh
2 bash getExamples.sh
```

This copies all examples into the `/workspace/examples` directory.

EXAMPLE LIST

The examples vary in complexity. It is recommended to use them in the following order:

1. [Mass Spring Damper](#)
2. [Water Tank Fault Injection](#)
3. [Water Tank Model Swap](#)
4. [Desktop Robotti and RabbitMQ](#)
5. [Water Treatment Plant and OPC-UA](#)
6. [Three Water Tanks with DT Manager Framework](#)
7. [Flex Cell with Two Industrial Robots](#)
8. [Incubator](#)
9. [Firefighters in Emergency Environments](#)
10. [Mass Spring Damper with NuRV Runtime Monitor FMU](#)
11. [Water Tank Fault Injection with NuRV Runtime Monitor FMU](#)
12. [Incubator Co-Simulation with NuRV Runtime Monitor FMU](#)
13. [Incubator with NuRV Runtime Monitor as Service](#)
14. [Incubator with NuRV Runtime Monitor FMU as Service](#)

 [DTaaS examples](#)

Mass Spring Damper

OVERVIEW

The mass spring damper digital twin (DT) comprises two mass spring dampers and demonstrates how a co-simulation based DT can be used within the DTaaS.

EXAMPLE DIAGRAM

Mass Spring Damper System

EXAMPLE STRUCTURE

There are two simulators included in the study, each representing a mass spring damper system. The first simulator calculates the mass displacement and speed of for a given force acting on mass . The second simulator calculates force given a displacement and speed of mass . By coupling these simulators, the evolution of the position of the two masses is computed.

Mass Spring Damper Structure

DIGITAL TWIN CONFIGURATION

This example uses two models and one tool. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Models	MassSpringDamper1.fmu	Private	Yes
	MassSpringDamper2.fmu	Private	Yes
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes

The `co-sim.json` and `time.json` are two DT configuration files used for executing the digital twin. These two files can be modified to customise the DT for specific requirements.

LIFECYCLE PHASES

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool
Execute	Produces and stores output in <code>data/mass-spring-damper/output</code> directory
Clean	Clears run logs and outputs

RUN THE EXAMPLE

To run the example, change the present directory.

```
1 cd /workspace/examples/digital_twins/mass-spring-damper
```

If required, change the execute permission of lifecycle scripts that need to be executed, for example:

```
1 chmod +x lifecycle/create
```

Now, run the following scripts:

Create

Installs Open Java Development Kit 17 in the workspace.

```
1 lifecycle/create
```

Execute

Run the the Digital Twin. Since this is a co-simulation based digital twin, the Maestro co-simulation tool executes co-simulation using the two FMU models.

```
1 lifecycle/execute
```

Examine the Results

The results can be found in the */workspace/examples/data/mass-spring-damper/output directory*.

Run logs can also be viewed in the */workspace/examples/digital_twins/mass-spring-damper*.

Terminate phase

Terminate to clean up the debug files and co-simulation output files.

```
1 lifecycle/terminate
```

REFERENCES

More information about co-simulation techniques and mass spring damper case study are available in:

- 1 Gomes, Cláudio, et al. "Co-simulation: State of the art."
- 2 arXiv preprint arXiv:1702.00686 (2017).

The source code for the models used in this DT are available in [mass spring damper](#) github repository.

Water Tank Fault Injection

OVERVIEW

This example demonstrates a fault injection (FI) enabled digital twin (DT). A live DT is subjected to simulated faults received from the environment. The simulated faults are specified as part of DT configuration and can be changed for new instances of DTs.

In this co-simulation based DT, a watertank case-study is used; co-simulation consists of a tank and controller. The goal of which is to keep the level of water in the tank between `Level-1` and `Level-2`. The faults are injected into output of the water tank controller (**Watertankcontroller-c.fmu**) from 12 to 20 time units, such that the tank output is closed for a period of time, leading to the water level increasing in the tank beyond the desired level (`Level-2`).

EXAMPLE DIAGRAM

Water Tank System

EXAMPLE STRUCTURE

Water Tank Structure

DIGITAL TWIN CONFIGURATION

This example uses two models and one tool. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Models	watertankcontroller-c.fmu	Private	Yes
	singlewatertank-20sim.fmu	Private	Yes
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes

The `multimodelFI.json` and `simulation-config.json` are two DT configuration files used for executing the digital twin. These two files can be modified to customise the DT for specific requirements.

 The faults are defined in **wt_fault.xml**.

LIFECYCLE PHASES

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool
Execute	Produces and stores output in <code>data/water_tank_FI/output</code> directory
Clean	Clears run logs and outputs

RUN THE EXAMPLE

To run the example, change the present directory.

```
1 cd /workspace/examples/digital_twins/water_tank_FI
```

If required, change the execute permission of lifecycle scripts that need to be executed, for example:

```
1 chmod +x lifecycle/create
```

Now, run the following scripts:

Create

Installs Open Java Development Kit 17 and pip dependencies. The pandas and matplotlib are the pip dependencies installed.

```
1 lifecycle/create
```

Execute

Run the co-simulation. Generates the co-simulation output.csv file at `/workspace/examples/data/water_tank_FI/output` .

```
1 lifecycle/execute
```

Analyze phase

Process the output of co-simulation to produce a plot at: `/workspace/examples/data/water_tank_FI/output/plots/` .

```
1 lifecycle/analyze
```

Examine the results

The results can be found in the `/workspace/examples/data/water_tank_FI/output` directory.

Run logs can also be viewed in the `/workspace/examples/digital_twins/water_tank_FI`.

Terminate phase

Clean up the temporary files and delete output plot

```
1 lifecycle/terminate
```

REFERENCES

More details on this case-study can be found in the paper:

```
1 M. Frasherì, C. Thule, H. D. Macedo, K. Lausdahl, P. G. Larsen and
2 L. Esterle, "Fault Injecting Co-simulations for Safety,"
3 2021 5th International Conference on System Reliability and Safety (ICSRS),
4 Palermo, Italy, 2021.
```

The fault-injection plugin is an extension to the Maestro co-orchestration engine that enables injecting inputs and outputs of FMUs in an FMI-based co-simulation with tampered values. More details on the plugin can be found in [fault injection](#) git repository. The source code for this example is also in the same github repository in a [example directory](#).

Water Tank Model Swap

OVERVIEW

This example demonstrates multi-stage execution and dynamic reconfiguration of a digital twin (DT). Two features of DTs are demonstrated:

- Fault injection into live DT
- Dynamic auto-reconfiguration of live DT

The co-simulation methodology is used to construct this DT.

EXAMPLE STRUCTURE

FMI Swap Structure

CONFIGURATION OF ASSETS

This example uses four models and one tool. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Models	Watertankcontroller-c.fmu	Private	Yes
	Singlewatertank-20sim.fmu	Private	Yes
	Leak_detector.fmu	Private	No
	Leak_controller.fmu	Private	No
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes

This DT has many configuration files. The DT is executed in two stages. There exist separate DT configuration files for each stage. The following table shows the configuration files and their purpose.

Configuration file name	Execution Stage	Purpose
mm1.json	stage-1	DT configuration
wt_fault.xml, FaultInject.mabl	stage-1	faults injected into DT during stage-1
mm2.json	stage-2	DT configuration
simulation-config.json	Both stages	Configuration for specifying DT execution time and output logs

LIFECYCLE PHASES

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool
Execute	Produces and stores output in data/water_tank_swap/output directory
Analyze	Process the co-simulation output and produce plots
Clean	Clears run logs, outputs and plots

RUN THE EXAMPLE

To run the example, change the present directory.

```
1 cd /workspace/examples/digital_twins/water_tank_swap
```

If required, change the permission of files that need to be executed, for example:

```
1 chmod +x lifecycle/create
```

Now, run the following scripts:

Create

Installs Open Java Development Kit 17 and pip dependencies. The matplotlib pip package is also installed.

```
1 lifecycle/create
```

Execute

This DT has two-stage execution. In the first-stage, a co-simulation is executed. The Watertankcontroller-c.fmu and Singlewatertank-20sim.fmu models are used to execute the DT. During this stage, faults are injected into one of the models (Watertankcontroller-c.fmu) and the system performance is checked.

In the second-stage, another co-simulation is run in which three FMUs are used. The FMUs used are: watertankcontroller, singlewatertank-20sim, and leak_detector. There is an in-built monitor in the Maestro tool. This monitor is enabled during the stage and a swap condition is set at the beginning of the second-stage. When the swap condition is satisfied, the Maestro swaps out Watertankcontroller-c.fmu model and swaps in Leakcontroller.fmu model. This swapping of FMU models demonstrates the dynamic reconfiguration of a DT.

The end of execution phase generates the co-simulation output.csv file at `/workspace/examples/data/water_tank_swap/output`.

```
1 lifecycle/execute
```

Analyze phase

Process the output of co-simulation to produce a plot at: `/workspace/examples/data/water_tank_FI/output/plots/`.

```
1 lifecycle/analyze
```

Examine the results

The results can be found in the `workspace/examples/data/water_tank_swap/output directory`.

Run logs can also be viewed in the `workspace/examples/digital_twins/water_tank_swap`.

Terminate phase

Clean up the temporary files and delete output plot

```
1 lifecycle/terminate
```

REFERENCES

The complete source of this example is available on [model swap](#) github repository.

The runtime model (FMU) swap mechanism demonstrated by the experiment is detailed in the paper:

```
1 Ejersbo, Henrik, et al. "fmiSwap: Run-time Swapping of Models for
2 Co-simulation and Digital Twins." arXiv preprint arXiv:2304.07328 (2023).
```

The runtime reconfiguration of co-simulation by modifying the Functional Mockup Units (FMUs) used is further detailed in the paper:

```
1 Ejersbo, Henrik, et al. "Dynamic Runtime Integration of
2 New Models in Digital Twins." 2023 IEEE/ACM 18th Symposium on
3 Software Engineering for Adaptive and Self-Managing Systems
4 (SEAMS). IEEE, 2023.
```

Desktop Robotti with RabbitMQ

OVERVIEW

This example demonstrates bidirectional communication between a mock physical twin and a digital twin of a mobile robot (Desktop Robotti). The communication is enabled by RabbitMQ Broker.

Desktop Robotti with RabbitMQ Broker and Digital Twin

EXAMPLE STRUCTURE

The mock physical twin of mobile robot is created using two python scripts

1. data/drobotti_rmcfmu/rmq-publisher.py
2. data/drobotti_rmcfmu/consume.py

The mock physical twin sends its physical location in (x,y) coordinates and expects a cartesian distance calculated from digital twin.

Desktop Robotti DT structure

The *rmq-publisher.py* reads the recorded (x,y) physical coordinates of mobile robot. The recorded values are stored in a data file. These (x,y) values are published to RabbitMQ Broker. The published (x,y) values are consumed by the digital twin.

The *consume.py* subscribes to RabbitMQ Broker and waits for the calculated distance value from the digital twin.

The digital twin consists of a FMI-based co-simulation, where Maestro is used as co-orchestration engine. In this case, the co-simulation is created by using two FMUs - RMQ FMU (rabbitmq-vhost.fmu) and distance FMU (distance-from-zero.fmu). The RMQ FMU receives the (x,y) coordinates from *rmq-publisher.py* and sends calculated distance value to *consume.py*. The RMQ FMU uses RabbitMQ broker for communication with the mock mobile robot, i.e., *rmq-publisher.py* and *consume.py*. The distance FMU is responsible for calculating the distance between $(0,0)$ and (x,y) . The RMQ FMU and distance FMU exchange values during co-simulation.

DIGITAL TWIN CONFIGURATION

This example uses two models, one tool, one data, and two scripts to create mock physical twin. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Models	distance-from-zero.fmu	Private	No
	rmq-vhost.fmu	Private	Yes
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes
Data	drobotti_playback_data.csv	private	No
Mock PT	rmq-publisher.py	Private	No
	consume.py	Private	No

This DT has many configuration files. The `coe.json` and `multimodel.json` are two DT configuration files used for executing the digital twin. These two files can be modified to customise the DT for specific requirements.

The RabbitMQ access credentials need to be provided in `multimodel.json`. The `rabbitmq-credentials.json` provides RabbitMQ access credentials for mock PT python scripts. The appropriate credentials should be added in both these files.

LIFECYCLE PHASES

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool and pip packages for python scripts
Execute	Runs both DT and mock PT
Clean	Clears run logs and outputs

RUN THE EXAMPLE

To run the example, change the present directory.

```
1 cd /workspace/examples/digital_twins/drobotti_rmcfmu
```

If required, change the execute permission of lifecycle scripts that need to be executed, for example:

```
1 chmod +x lifecycle/create
```

Now, run the following scripts:

Create

Installs Open Java Development Kit 17 in the workspace. Also install the required python pip packages for *rmq-publisher.py* and *consume.py* scripts.

```
1 lifecycle/create
```

Execute

Run the python scripts to start mock physical twin. Also run the the Digital Twin. Since this is a co-simulation based digital twin, the Maestro co-simulation tool executes co-simulation using the two FMU models.

```
1 lifecycle/execute
```

Examine the results

The results can be found in the */workspace/examples/digital_twins/drobotti_rmcfmu directory*.

Executing the DT will generate and launch a co-simulation (RMQFMU and distance FMU), and two python scripts. One to publish data that is read from a file. And one to consume what is sent by the distance FMU.

In this examples the DT will run for 10 seconds, with a stepsize of 100ms. Thereafter it is possible to examine the logs produce in */workspace/examples/digital_twins/drobotti_rmcfmu/target* . The outputs for each FMU, xpos and ypos for the RMQFMU, and the distance for the distance FMU are recorded in the *outputs.csv* file. Other logs can be examined for each FMU and the publisher scripts. Note that, the RMQFMU only sends data, if the current input is different to the previous one.

Terminate phase

Terminate to clean up the debug files and co-simulation output files.

```
1 lifecycle/terminate
```

REFERENCES

The [RabbitMQ FMU](#) github repository contains complete documentation and source code of the *rmq-vhost.fmu*.

More information about the case study is available in:

1

Fraseri, Mirgita, et al. "Addressing time discrepancy between digital and physical twins." *Robotics and Autonomous Systems* 161 (2023): 104347.

2

Waste Water Plant with OPC-UA

INTRODUCTION

Waste water treatment (WWT) plants must comply with substance and species concentration limits established by regulation in order to ensure the good quality of the water. This is usually done taking periodic samples that are analysed in the laboratory. This means that plant operators do not have continuous information for making decisions and, therefore, operation setpoints are set to higher values than needed to guarantee water quality. Some of the processes involved in WWT plants consume a lot of power, thus adjusting setpoints could significantly reduce energy consumption.

PHYSICAL TWIN OVERVIEW

This example demonstrates the communication between a physical ultraviolet (UV) disinfection process (the tertiary treatment of a WWT plant) and its digital twin, which is based on Computational Fluid Dynamics (CFD) and compartment models. The aim of this digital twin is to develop "virtual sensors" that provide continuous information that facilitates the decision making process for the plant operator.

Waste water treatment plant

The physical twin of the waste water plant is composed of an ultraviolet channel controlled by a PLC that controls the power of the UV lamps needed to kill all the pathogens of the flow. The channel has 3 groups of UV lamps, therefore the real channel (and its mathematical model) is subdivided into 7 zones: 4 correspond to zones without UV lamps (2 for the entrance and exit of the channel + 2 zones between UV lamps) and the 3 remaining for the UV lamps.

The dose to be applied (related with the power) changes according to the residence time (computed from the measure of the volume flow) and the UV intensity (measured by the intensity sensor).

The information of the volumetric flow and power (in the three parts of the channel) is transmitted to the PLC of the plant. Furthermore, the PLC is working as OPC UA Server to send and receive data to and from an OPC UA Client. Additionally, some sizing parameters and initial values are read from a spreadsheet filled in by the plant operator. In this case, the spreadsheet is an Open Office file (.ods) due to the software installed in the SCADA PC. Some of the variables like initial concentration of disinfectant and pathogens are included, among others. Some values defined by the plant operator correspond to input signals that are not currently being measured, but are expected to be measured in the future.

DIGITAL TWIN OVERVIEW

Digital Twin Reduced Order Model

The digital twin is a reduced model (developed in C) that solves physical conservation laws (mass, energy and momentum), but simplifies details (geometry, mainly) to ensure real-time calculations and accurate results. The results are compared to the ones obtained by the CFD. C solver developed is used by the OpenModelica model. OpenModelica converts it into the FMI standard, to be integrated in the OPC UA Client (*client-opcua.py*).

Structure of Python Script

DIGITAL TWIN CONFIGURATION

Asset Type	Name of Asset	Visibility	Reuse in Other Examples
Model	Test_DTCONEDAR.mo	private	No
Data	configuration_freopcua.ods	private	No
	model_description.csv (generated by client-asyncua.py)	private	No
	Mock OPC UA Server: opcua-mock-server.py	private	No
Tool	OPC UA Client: client-opcua.py	private	No
	FMU builder: create-fmu.mos	private	No

In this example, a dummy model representation of the plant is used, instead of the real model. The simplified model (with not the real equations) is developed in **Open Modelica (Test_DTCONEDAR.mo)**. The FMU is generated from the Open Modelica interface to obtain the needed binaries to run the FMU. It is possible to run an FMU previously generated, however, to ensure that the right binaries are being used, it is recommended to install Open Modelica Compiler and run `script.mos` to build the FMU from the Modelica file `Test_DTCONEDAR.mo`.

The FMU model description file (`modelDescription.xml` file inside `Test_DTCONEDAR.fmu`) has the information of the value references `configuration_freeopcua.ods` has the information of the OPC-UA node IDs And both have in common the variable name

The client python script (`client-opcua.py`) does the following actions:

- Reads the variable names and the variable value references from the model description file of the `Test_DTCONEDAR.FMU`.
- Reads `configuration_freeopcua.ods` to obtain opcua node IDs and assigns those node IDs to the variables read from the FMU
- Read `configuration_freeopcua.ods` to fix initial values, parameters and some inputs (those inputs that are not being measured, a reasonable value is assumed).
- Read values from PLC using a client OPC.
- Execute the algorithm with the FMPy library using the `.fmu` created from the compartment model (based on CFD)
- Obtain results.
- Send by OPC UA protocol the result values to the PLC, to visualise them in the SCADA and with the aim to improve the decision-making process of the plant operator.

Input Data Variables

The `configuration_freeopcua.ods` data file is used for customising the initial input data values used by the server.

Configuration parameters

Configuration input

DT Config

The `config.json` specifies the configuration parameters for the OPC UA client.

Config .JSON

Optional parameters can be modified:

- `stop_time`
- `step_size`
- `record = True`, to save the results of the simulation
- `record_interval`. Sometimes the simulation `step_size` is small and a the size of the results file can be too big. For instance, if the simulation `step_size` is 0.01 seconds, the `record_interval` can be increased so as to reduce the result file size.
- `record_variables`: the list of variables to record can be specified.
- `enable_send = True`, to send results to the OPC UA Server.

LIFECYCLE PHASES

The lifecycles that are covered include:

Lifecycle Phase	Completed Tasks
Install	Installs Open Modelica, Python 3.10 and the required pip dependencies
Create	Create FMU from Open Modelica file
Execute	Run OPC UA mock server and normal OPC-UA client
Clean	Delete the temporary files

RUN THE EXAMPLE

To run the example, change the present directory.

```
1 cd /workspace/examples/digital_twins/opc-ua-waterplant
```

If required, change the execute permission of lifecycle scripts.

```
1 chmod +x lifecycle/*
```

Now, run the following scripts:

Install

Installs Open Modelica, Python 3.10 and the required pip dependencies

```
1 lifecycle/install
```

Create

Create `Test_DTCONEDAR.fmu` co-simulation model from `Test_DTCONEDAR.mo` open modelica file.

```
1 lifecycle/create
```

Execute

Start the mock OPC UA server in the background. Run the OPC UA client.

```
1 lifecycle/execute
```

Clean

Remove the temporary files created by Open Modelica and output files generated by OPC UA client.

```
1 lifecycle/clean
```

REFERENCES

More explanation about this example is available at:

```
1 Royo, L., Labarías, A., Arnau, R., Gómez, A., Ezquerro, A., Cilla, I., &
2 Díez-Antoñanzas, L. (2023). Improving energy efficiency in the tertiary
3 treatment of Alguazas WWTP (Spain) by means of digital twin development
4 based on Physics modelling . Cambridge Open Engage.
5 [doi:10.33774/coe-2023-1vjcw](https://doi.org/10.33774/coe-2023-1vjcw)
```

ACKNOWLEDGEMENTS

The work on this example was done in a project subsidised thanks to the support grants for Agrupación Empresarial Innovadora (AEI) of the Ministry of Industry, Trade and Tourism (MINCOTUR) with the aim of improving the competitiveness of small and medium-sized enterprises within the Recovery, Transformation and Resilience Plan (PRTR) financed by the Next Generation EU (NGEU) funds, with Grant Number: AEI-010500-2022b-196.

Three-Tank System Digital Twin

OVERVIEW

The three-tank system is a simple case study that represents a system composed of three individual components coupled in a cascade as follows: The first tank is connected to the input of the second tank, and the output of the second tank is connected to the input of the third tank.

Three-tank graphical representation

This example contains only the simulated components for demonstration purposes; therefore, there is no configuration for the connection with the physical system.

The three-tank system case study is managed using the `DTManager`, which is packed as a jar library in the tools, and run from a java main file. The `DTManager` uses Maestro as a slave for co-simulation, so it generates the output of the co-simulation.

The main file can be changed according to the application scope, i.e., the `/workspace/examples/tools/three-tank/TankMain.java` can be manipulated to get a different result.

The `/workspace/examples/models/three-tank/` folder contains the `Linear.fmu` file, which is a non-realistic model for a tank with input and output and the `TankSystem.aasx` file for the schema representation with Asset Administration Shell. The three instances use the same `.fmu` file and the same schema due to being of the same object class. The `DTManager` is in charge of reading the values from the co-simulation output.

EXAMPLE STRUCTURE

Three-tank system architecture with DT Manager

DIGITAL TWIN CONFIGURATION

This example uses two models, two tools, one data, and one script. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Model	Linear.fmu	Private	No
	TankSystem.aasx	Private	No
Tool	DTManager-0.0.1-Maestro.jar (wraps Maestro)	Common	Yes
	maestro-2.3.0-jar-with-dependencies.jar (used by DTManager)	Common	Yes
	TankMain.java (main script)	Private	No
Data	outputs.csv	Private	No

This DT has multiple configuration files. The `coe.json` and `multimodel.json` are used by Maestro tool. The `tank1.conf`, `tank2.conf` and `tank3.conf` are the config files for three different instances of one model (Linear.fmu).

LIFECYCLE PHASES

The lifecycles that are covered include:

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool
Execute	The DT Manager executes the three-tank digital twin and produces output in <code>data/three-tank/output</code> directory
Terminate	Terminating the background processes and cleaning up the output

RUN THE EXAMPLE

To run the example, change the present directory.

```
1 cd /workspace/examples/digital_twins/three-tank
```

If required, change the execute permission of lifecycle scripts that need to be executed, for example:

```
1 chmod +x lifecycle/create
```

Now, run the following scripts:

Create

Installs Open Java Development Kit 11 and pip dependencies. Also creates `DTManager` tool (DTManager-0.0.1-Maestro.jar) from source code.

```
1 lifecycle/create
```

Execute

Execute the three-tank digital twin using DTManager. DTManager in-turn runs the co-simulation using Maestro. Generates the co-simulation output.csv file at `/workspace/examples/data/three-tank/output`.

```
1 lifecycle/execute
```

Terminate

Stops the Maestro running in the background. Also stops any other jvm process started during **execute** phase.

```
1 lifecycle/terminate
```

Clean

Removes the output generated during execute phase.

```
1 lifecycle/terminate
```

EXAMINING THE RESULTS

Executing this Digital Twin will generate a co-simulation output, but the results can also be monitored from updating the `/workspace/examples/tools/three-tank/TankMain.java` with a specific set of `getAttributeValue` commands, such as shown in the code.

That main file enables the online execution of the Digital Twin and its internal components.

The output of the co-simulation is generated to the `/workspace/examples/data/three-tank/output` folder.

In the default example, the co-simulation is run for 10 seconds in steps of 0.5 seconds. This can be modified for a longer period and different step size. The output stored in `outputs.csv` contains the level, in/out flow, and leak values.

No data from the physical twin are generated/used.

REFERENCES

More information about the DT Manager is available at:

```
1 D. Lehner, S. Gil, P. H. Mikkelsen, P. G. Larsen and M. Wimmer,
2 "An Architectural Extension for Digital Twin Platforms to Leverage
3 Behavioral Models," 2023 IEEE 19th International Conference on
4 Automation Science and Engineering (CASE), Auckland, New Zealand,
5 2023, pp. 1-8, doi: 10.1109/CASE56687.2023.10260417.
```

Flex Cell Digital Twin with Two Industrial Robots

OVERVIEW

The flex-cell Digital Twin is a case study with two industrial robotic arms, a UR5e and a Kuka LBR iiwa 7, working in a cooperative setting on a manufacturing cell.

Flex-cell Physical Setup

The case study focuses on the robot positioning in the discrete cartesian space of the flex-cell working space. Therefore, it is possible to send (X,Y,Z) commands to both robots, which refer to the target hole and height to which they should move.

The flex-cell case study is managed using the `TwinManager` (formerly `DT Manager`), which is packed as a jar library in the tools, and run from a java main file.

The `TwinManager` uses Maestro as a slave for co-simulation, so it generates the output of the co-simulation and can interact with the real robots at the same time (with the proper configuration and setup). The mainfile can be changed according to the application scope, i.e., the `/workspace/examples/tools/flex-cell/FlexCellDTaaS.java` can be manipulated to get a different result.

The `/workspace/examples/models/flex-cell/` folder contains the `.fmu` files for the kinematic models of the robotic arms, the `.urdf` files for visualisation (including the grippers), and the `.aasx` files for the schema representation with Asset Administration Shell.

The case study also uses RabbitMQFMU to inject values into the co-simulation, therefore, there is the `rabbitmqfmu` in the models folder as well. Right now, RabbitMQFMU is only used for injecting values into the co-simulation, but not the other way around. The `TwinManager` is in charge of reading the values from the co-simulation output and the current state of the physical twins.

EXAMPLE STRUCTURE

The example structure represents the components of the flex-cell DT implementation using the `TwinManager` architecture.

The `TwinManager` orchestrates the flex-cell DT via the *Flex-cell DT System*, which is composed of 2 smaller Digital Twins, namely, the *DT UR5e* and the *DT Kuka lbr iiwa 7*. The `TwinManager` also provides the interface for the Physical Twins, namely, *PT UR5e* and *PT Kuka lbr iiwa 7*. Each Physical Twin and Digital Twin System has a particular endpoint (with a different specialization), which is initialized from configuration files and data model (twin schema).

The current endpoints used in this implementation are:

Digital or Physical Twin	Endpoint
Flex-cell DT System	MaestroEndpoint
DT UR5e	FMIEndpoint
DT Kuka lbr iiwa 7	FMIEndpoint
PT UR5e	MQTTEndpoint and RabbitMQEndpoint
PT Kuka lbr iiwa 7	MQTTEndpoint and RabbitMQEndpoint

The Flex-cell DT System uses another configuration to be integrated with the Maestro co-simulation engine.

In the lower part, the Flex-cell System represents the composed physical twin, including the two robotic arms and controller and the Flex-cell Simulation is the mock-up representation for the real system, which is implemented by FMU blocks and their connections.

Flex-cell system architecture with the `TwinManager`

Internals of PT and DT

DIGITAL TWIN CONFIGURATION

This example uses seven models, five tools, six data files, two functions, and one script. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Model	kukalbriiwa_model.fmu	Private	No
	kuka_irw_gripper_rg6.urdf	Private	No
	kuka.aasx	Private	No
	ur5e_model.fmu	Private	No
	ur5e_gripper_2fg7.urdf	Private	No
	ur5e.aasx	Private	No
	rmq-vhost.fmu	Private	Yes
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes
	TwinManagerFramework-0.0.2.jar	Private	Yes
	urinterface (installed with pip)	Private	No
	kukalbrinterface	Private	No
	robots_flexcell	Private	No
	FlexCellDTaaS.java (main script)	Private	No
Data	publisher-flexcell-physical.py	Private	No
	ur5e_mqtt_publisher.py	Private	No
	connections.conf	Private	No
	outputs.csv	Private	No
	kukalbriiwa7_actual.csv	Private	No
	ur5e_actual.csv	Private	No
Function	plots.py	Private	No
	prepare.py	Private	No

LIFECYCLE PHASES

The lifecycles that are covered include:

1. Installation of dependencies in the create phase.
2. Preparing the credentials for connections in the prepare phase.
3. Execution of the experiment in the execution phase.
4. Saving experiments in the save phase.
5. Plotting the results of the co-simulation and the real data coming from the robots in the analyse phase.
6. Terminating the background processes and cleaning up the outputs in the termination phase.

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool, Compiles source code of TwinManager to create a usable jar package (used as tool)
Prepare	Takes the RabbitMQ and MQTT credentials in connections.conf file and configures different assets of DT.
Execute	The TwinManager executes the flex-cell DT and produces output in <code>data/flex-cell/output</code> directory
Save	Save the experimental results
Analyze	Uses plotting functions to generate plots of co-simulation results
Terminate	Terminating the background processes
Clean	Cleans up the output data

RUN THE EXAMPLE

To run the example, change the present directory.

```
1 cd /workspace/examples/digital_twins/flex-cell
```

If required, change the execute permission of lifecycle scripts that need to be executed, for example:

```
1 chmod +x lifecycle/create
```

This example requires Java 11. The **create** script installs Java 11; however if other Java versions have already been installed, the default *java* may be pointing to another version. The default version can be checked and modified using the following commands.

```
1 java -version
2 update-alternatives --config java
```

Now, run the following scripts:

Create

Installs Open Java Development Kit 11 and a python virtual environment with pip dependencies. Also builds the `TwinManager` tool (TwinManagerFramework-0.0.2.jar) from source code.

```
1 lifecycle/create
```

Prepare

This step configures different assets of the DT with connection credentials. The `functions/flex-cell/prepare.py` script is used for this purpose. The only step needed to set up the connection is to update the file `/workspace/examples/data/flex-cell/input/connections.conf` with the connection parameters for MQTT and RabbitMQ and then execute the `prepare` script.

```
1 lifecycle/prepare
```

The following files are updated with the configuration information:

1. `/workspace/examples/digital_twins/flex-cell/kuka_actual.conf`
2. `/workspace/examples/digital_twins/flex-cell/ur5e_actual.conf`
3. `/workspace/examples/data/flex-cell/input/publisher-flexcell-physical.py`
4. `modelDescription.xml` for the RabbitMQFMU require special credentials to connect to the RabbitMQ and the MQTT brokers.

Execute

Execute the flex-cell digital twin using TwinManager. TwinManager in-turn runs the co-simulation using Maestro. Generates the co-simulation output.csv file at `/workspace/examples/data/flex-cell/output`. The execution needs to be stopped with `control + c` since the `TwinManager` runs the application in a non-stopping loop.

```
1 lifecycle/execute
```

Save

Each execution of the DT is treated as a single run. The results of one execution are saved as time-stamped co-simulation output file in The TwinManager executes the flex-cell digital twin and produces output in `data/flex-cell/output/saved_experiments` directory.

```
1 lifecycle/save
```

The `execute` and `save` scripts can be executed in that order any number of times. A new file `data/flex-cell/output/saved_experiments` directory with each iteration.

Analyze

There are dedicated plotting functions in `functions/flex-cell/plots.py`. This script plots the co-simulation results against the recorded values from the two robots.

```
1 lifecycle/analyze
```

Terminate

Stops the Maestro running in the background. Also stops any other jvm process started during **execute** phase.

```
1 lifecycle/terminate
```

Clean

Removes the output generated during execute phase.

```
1 lifecycle/clean
```

EXAMINING THE RESULTS

Executing this Digital Twin generates a co-simulation output. The results can also be monitored by updating the `/workspace/examples/tools/flex-cell/FlexCellDTaaS.java` with a specific set of `getAttributeValue` commands, as shown in the code. That main file enables the online execution and comparison of Digital Twin and Physical Twin at the same time and at the same abstraction level.

The output is generated to the `/workspace/examples/data/flex-cell/output` folder. In case a specific experiments is to be saved, the **save** lifecycle script stores the co-simulation results into the `/workspace/examples/data/flex-cell/output/saved_experiments` folder.

In the default example, the co-simulation is run for 11 seconds in steps of 0.2 seconds. This can be modified for a longer period and different step size. The output stored in `outputs.csv` contains the joint position of both robotic arms and the current discrete (X,Y,Z) position of the TCP of the robot. Additional variables can be added, such as the discrete (X,Y,Z) position of the other joints.

When connected to the real robots, the tools `urinterface` and `kukalbrinterface` log their data at a higher sampling rate.

REFERENCES

The [RabbitMQ FMU](#) github repository contains complete documentation and source code of the **rmq-vhost.fmu**.

More information about the TwinManager (formerly DT Manager) and the case study is available in:

1. D. Lehner, S. Gil, P. H. Mikkelsen, P. G. Larsen and M. Wimmer, "An Architectural Extension for Digital Twin Platforms to Leverage Behavioral Models," 2023 IEEE 19th International Conference on Automation Science and Engineering (CASE), Auckland, New Zealand, 2023, pp. 1-8, doi: 10.1109/CASE56687.2023.10260417.
2. S. Gil, P. H. Mikkelsen, D. Tola, C. Schou and P. G. Larsen, "A Modeling Approach for Composed Digital Twins in Cooperative Systems," 2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA), Sinaia, Romania, 2023, pp. 1-8, doi: 10.1109/ETFA54631.2023.10275601.
3. S. Gil, C. Schou, P. H. Mikkelsen, and P. G. Larsen, "Integrating Skills into Digital Twins in Cooperative Systems," in 2024 IEEE/ SICE International Symposium on System Integration (SII), 2024, pp. 1124-1131, doi: 10.1109/SII58957.2024.10417610.

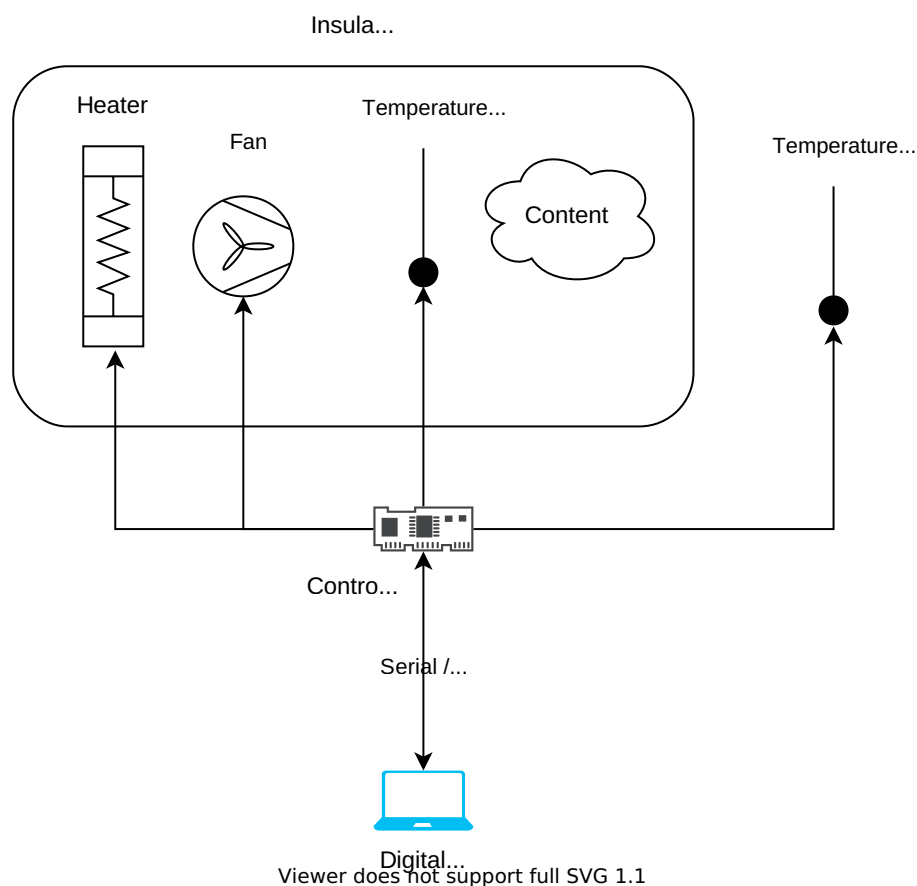
Incubator Digital Twin

OVERVIEW

This is a case study of an Incubator intended to demonstrate the steps and processes involved in developing a digital twin system. This incubator is an insulated container with the ability to maintain a temperature and provide heating, but not cooling. A picture of the incubator is provided below.

Incubator Picture

The overall purpose of the system is to reach a certain temperature within a box and keep the temperature regardless of content. An overview of the system can be seen below:



The system consists of:

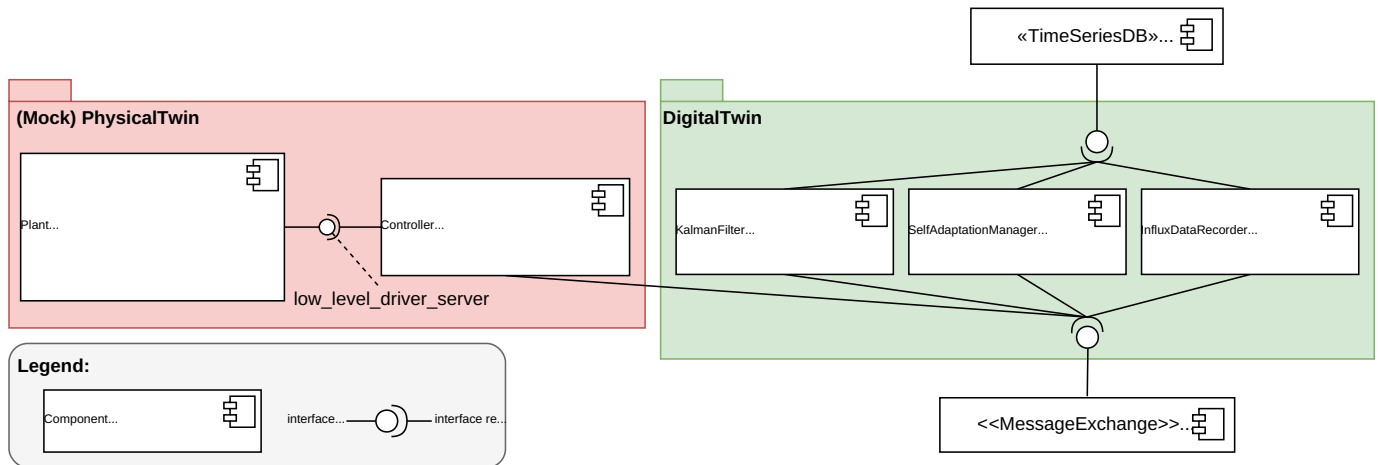
- 1x styrofoam box in order to have an insulated container
- 1x heat source to heat up the content within the Styrofoam box.
- 1x fan to distribute the heating within the box
- 2x temperature sensor to monitor the temperature within the box
- 1x temperature Sensor to monitor the temperature outside the box
- 1x controller to actuate the heat source and the fan and read sensory information from the temperature sensors, and communicate with the digital twin.

The original repository for the example can be found: [Original repository](#). This trimmed version of the codebase does not have the following:

- docker support
- tests
- datasets

The original repository contains the complete documentation of the example, including the full system architecture, instructions for running with a physical twin, and instructions for running a 3D visualisation of the incubator.

DIGITAL TWIN STRUCTURE



This diagram shows the main components and the interfaces they use to communicate. All components communicate via the RabbitMQ message exchange, and the data is stored in a time series database. The RabbitMQ and InfluxDB are platform services hosted by the DTaaS.

The Incubator digital twin is a pre-packaged digital twin. It can be used as is or integrated with other digital twins.

The mock physical twin is executed from `incubator/mock_plant/real_time_model_solver.py` script.

DIGITAL TWIN CONFIGURATION

This example uses a plethora of Python scripts to run the digital twin. By default, it is configured to run with a mock physical twin. Furthermore, it depends on RabbitMQ and InfluxDB instances.

There is one configuration file: `simulation.conf`. The RabbitMQ and InfluxDB configuration parameters must be updated.

LIFECYCLE PHASES

The lifecycles that are covered include:

Lifecycle Phase	Completed Tasks
Create	Potentially updates the system and installs Python dependencies
Execute	Executes the Incubator digital twin and produces output in the terminal and in <code>incubator/log.log</code> .
Clean	Removes the log file.

RUN THE EXAMPLE

To run the example, change the present directory.

```
1 cd /workspace/examples/digital_twins/incubator
```

If required, change the execute permission of lifecycle scripts that need to be executed, for example:

```
1 chmod +x lifecycle/create
```

Now, run the following scripts:

Create

Potentially updates the system and installs Python dependencies.

```
1 lifecycle/create
```

Execute

Executes the Incubator digital twin with a mock physical twin. Pushes the results in the terminal, *incubator/log.log*, and in InfluxDB.

```
1 lifecycle/execute
```

Clean

Removes the output log file.

```
1 lifecycle/clean
```

EXAMINING THE RESULTS

After starting all services successfully, the controller service will start producing output that looks like the following:

1	time	execution_interval	elapsed	heater_on	fan_on	room	box_air_temperature	state
2	19/11 16:17:59	3.00	0.01	True	False	10.70	19.68	Heating
3	19/11 16:18:02	3.00	0.03	True	True	10.70	19.57	Heating
4	19/11 16:18:05	3.00	0.01	True	True	10.70	19.57	Heating
5	19/11 16:18:08	3.00	0.01	True	True	10.69	19.47	Heating
6	19/11 16:18:11	3.00	0.01	True	True	10.69	19.41	Heating

An InfluxDB dashboard can be setup based on `incubator/digital_twin/data_access/influxdbserver/dashboards/incubator_data.json`.

If the dashboard on the InfluxDB is setup properly, the following visualisation can be seen:

Influx Dashboard

REFERENCES

 Forked from: [Incubator repository](#) with commit ID: 989ccf5909a684ad26a9c3ec16be2390667643aa

To understand what a digital twin is, one or more of the following resources are recommended:

1. Feng, Hao, Cláudio Gomes, Casper Thule, Kenneth Lausdahl, Alexandros Iosifidis, and Peter Gorm Larsen. "Introduction to Digital Twin Engineering." In 2021 Annual Modeling and Simulation Conference (ANNSIM), 1–12. Fairfax, VA, USA: IEEE, 2021. <https://doi.org/10.23919/ANNSIM52504.2021.9552135>.
2. [Video recording of presentation by Claudio Gomes](#)

Firefighter Mission in a Burning Building

In a firefighter mission, it is important to monitor the oxygen levels of each firefighter's Self Contained Breathing Apparatus (SCBA) in the context of their mission.

PHYSICAL TWIN OVERVIEW

Burning building

Image: Schematic overview of a firefighter mission. Note the mission commander on the lower left documenting the air supply pressure levels provided by radio communication from the firefighters inside and around the burning building. This image was created with the assistance of DALL·E.

The following scenario is assumed:

- a set of firefighters work to extinguish a burning building
- they each use an SCBA with pressurised oxygen to breath
- a mission commander on the outside coordinates the efforts and surveills the oxygen levels

DIGITAL TWIN OVERVIEW

In this example a monitor is implemented, that calculates how much time the firefighters have left, until they need to leave the building. To that end, the inputs used are:

- 3D-model of the building in which the mission takes place,
- pressure data of a firefighters SCBA and
- firefighters location inside of the building

are used to estimate:

- the shortest way out,
- how much time this will need and
- how much time is left until all oxygen from the SCBA is used up.

The remaining mission time is monitored and the firefighter receive a warning if it drops under a certain threshold.

Digital Twin Structure

This example is an implementation of the the paper [Digital Twin for Rescue Missions--a Case Study](#) by Leucker et al.

Quick Check

Before running this example, the following files must be verified to be at the correct locations:

```

1  /workspace/examples/
2    data/o5g/input/
3      runTesla.sh
4      sensorSimulation.py
5      telegraf.conf
6
7    models/
8      lab.ifc
9      makefmu.mos
10     PathOxygenEstimate.mo
11
12   tools/
13     graphToPath.py
14     ifc_to_graph
15     pathToTime.py
16     tessla-telegraf-connector/
17     tessla-telegraf-connector/
18     tessla.jar
19     specification.tessla (run-time specification)
20
21   digital_twins/o5g/
22     main.py
23     config
24     lifecycle/ (scripts)

```

Digital Twin Configuration

All configuration for this example is contained in `digital_twins/o5g/config`.

To use the MQTT-Server, account information needs to be provided. The topics are set to their default values, which allow the DT to access the mock physical twins sensor metrics and to send back alerts.

```
1 export O5G_MQTT_SERVER=
2 export O5G_MQTT_PORT=
3 export O5G_MQTT_USER=
4 export O5G_MQTT_PASS=
5
6 export O5G_MQTT_TOPIC_SENSOR='vgiot/ue/metric'
7 export O5G_MQTT_TOPIC_AIR_PREDICTION='vgiot/dt/prediction'
8 export O5G_MQTT_TOPIC_ALERT='vgiot/dt/alerts'
```

This example uses InfluxDB as a data storage, which will need to be configured to use the relevant access data. The following configuration steps are needed:

- Log into the InfluxDB Web UI
- Obtain **org** name (below the *username* in the sidebar)
- Create a data bucket if one does not already exist in `Load Data -> Buckets`
- Create an API access token in `Load Data -> API Tokens`, Copy and save this token immediately; it cannot be accessed subsequently.

```
1 export O5G_INFLUX_SERVER=
2 export O5G_INFLUX_PORT=
3 export O5G_INFLUX_TOKEN=
4 export O5G_INFLUX_ORG=
5 export O5G_INFLUX_BUCKET=
```

LIFECYCLE PHASES

The lifecycles that are covered include:

Lifecycle Phase	Completed Tasks
Install	Installs Open Modelica, Rust, Telegraf and the required pip dependencies
Create	Create FMU from Open Modelica file
Execute	Execute the example in the background tmux terminal session
Terminate	Terminate the tmux terminal session running in the background
Clean	Delete the temporary files

RUN THE EXAMPLE

Install

Run the install script by executing

```
1 lifecycle/install
```

This will install all the required dependencies from apt and pip, as well as Open Modelica, Rust, Telegraf and the required pip dependencies from their respective repos.

Create

Run the create script by executing

```
1 lifecycle/create
```

This will compile the modelica model to an Functional Mockup Unit (FMU) for the correct platform.

Execute

To run the Digital Twin execute

```
1 lifecycle/execute
```

This will start all the required components in a single `tmux` session called `o5g` in the background. To view the running Digital Twin attach to this `tmux` session by executing

```
1 tmux a -t o5g
```

To detach press `ctrl-b` followed by `d`.

Tmux session of live digital twin

The `tmux` session contains 4 components of the digital twin:

Panel location	Purpose
Top Left	Sensor simulator generating random location and O2-level data
Top Right	Main Digital Twin receives the sensor data and calculates an estimate of how many minutes of air remain
Bottom Left	Telegraf to convert between different message formats, also displays all messages between components
Bottom Right	TeSSLa monitor raises an alarm, if the remaining time is to low.

Examine the Results

For additional mission awareness, it is recommended to utilise the Influx data visualisation. A dashboard configuration is provided in the file `influx-dashoard.json`. Log in to the Influx Server to import (usually port 8086). A screenshot of the dashboard is given here.

Firefighter remaining mission time

The data gets stored in `o5g->prediction->air-remaining->37ae3e4fb3ea->true->vgiot/dt/prediction` variable of the InfluxDB. In addition to importing dashboard configuration given above, it is possible to create custom dashboards using the stored data.

Terminate

To stop the all components and close the `tmux` session execute

```
1 lifecycle/terminate
```

Clean

To remove temporary files created during execution

```
1 lifecycle/clean
```

Mass Spring Damper with NuRV Runtime Monitor

OVERVIEW

This digital twin is derived from the Mass Spring Damper digital twin.

The mass spring damper digital twin (DT) comprises two mass spring dampers and demonstrates how a co-simulation based DT can be used within the DTaaS. This version of the example is expanded with a monitor generated by NuRV. More information about NuRV is available on the [NuRV tool page](#).

EXAMPLE DIAGRAM

Mass Spring Damper System

EXAMPLE STRUCTURE

There are two simulators included in the study, each representing a mass spring damper system. The first simulator calculates the mass displacement and speed of for a given force acting on mass . The second simulator calculates force given a displacement and speed of mass . By coupling these simulators, the evolution of the position of the two masses is computed.

Additionally, a monitor is inserted in the simulation to check at runtime whether the displacement of the two masses stays below a maximum threshold.

Mass Spring Damper Structure

DIGITAL TWIN CONFIGURATION

This example uses two models and one tool. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Models	MassSpringDamper1.fmu	Private	Yes
	MassSpringDamper2.fmu	Private	Yes
	m2.fmu	Private	No
	RtI.fmu	Private	Yes
Specification	m2.smv	Private	No
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes

The `co-sim.json` and `time.json` are two DT configuration files used for executing the digital twin. These two files can be modified to customise the DT for specific requirements.

LIFECYCLE PHASES

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool Generates and compiles the monitor FMU
Execute	Produces and stores output in data/mass-spring-damper-monitor/output directory
Clean	Clears run logs and outputs

RUN THE EXAMPLE

To run the example, change the present directory.

```
1 cd /workspace/examples/digital_twins/mass-spring-damper-monitor
```

If required, change the execute permission of lifecycle scripts that need to be executed, for example:

```
1 chmod +x lifecycle/create
```

Now, run the following scripts:

Create

- Installs Open Java Development Kit 17 in the workspace.
- Generates and compiles the monitor FMU from the NuRV specification

```
1 lifecycle/create
```

Execute

Run the the Digital Twin. Since this is a co-simulation based digital twin, the Maestro co-simulation tool executes co-simulation using the two FMU models.

```
1 lifecycle/execute
```

Analyze phase

Process the output of co-simulation to produce a plot at: `/workspace/examples/data/mass-spring-damper-monitor/output/plots`.

```
1 lifecycle/analyze
```

A sample plot is given here.

sample plot

In the plot, three colour-coded indicators are used to represent different values. The blue line shows the distance between the two masses, while the green indicates the monitor's verdict. A red dashed line serves as a reference point, marking the distance checked by the monitor. Since the distance of the masses is always below the threshold, the output of the monitor is fixed to `unknown (0)`.

Examine the results

The results can be found in the `/workspace/examples/data/mass-spring-damper-monitor/output` directory.

Run logs can also be viewed in the `/workspace/examples/digital_twins/mass-spring-damper-monitor`.

Terminate phase

Terminate to clean up the debug files and co-simulation output files.

```
1 lifecycle/terminate
```

REFERENCES

More information about co-simulation techniques and mass spring damper case study are available in:

```
1 Gomes, Cláudio, et al. "Co-simulation: State of the art."
2 arXiv preprint arXiv:1702.00686 (2017).
```

The source code for the models used in this DT are available in [mass spring damper](#) github repository.

Water Tank Fault Injection with NuRV Runtime Monitor

OVERVIEW

This example demonstrates a fault injection (FI) enabled digital twin. A live DT is subjected to simulated faults received from the environment. The simulated faults are specified as part of DT configuration and can be changed for new instances of DTs. This version of the example is expanded with a monitor generated by NuRV. More information about NuRV is available on the [NuRV tool page](#).

In this co-simulation based DT, a watertank case-study is used; co-simulation consists of a tank and controller. The goal of which is to keep the level of water in the tank between `Level-1` and `Level-2`. The faults are injected into output of the water tank controller (**Watertankcontroller-c.fmu**) from 12 to 20 time units, such that the tank output is closed for a period of time, leading to the water level increasing in the tank beyond the desired level (`Level-2`). Additionally, a monitor is inserted in the simulation to check at runtime whether the level of the water stays below a maximum threshold.

EXAMPLE DIAGRAM

Water Tank System

EXAMPLE STRUCTURE

Water Tank Structure

DIGITAL TWIN CONFIGURATION

This example uses two models and one tool. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Models	watertankcontroller-c.fmu	Private	Yes
	singlewatertank-20sim.fmu	Private	Yes
	m1.fmu	Private	No
	RtI.fmu	Private	Yes
Specification	m1.smv	Private	No
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes

The `multimodelFI.json` and `simulation-config.json` are two DT configuration files used for executing the digital twin. These two files can be modified to customise the DT for specific requirements.

 The faults are defined in **wt_fault.xml**.

LIFECYCLE PHASES

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool Generates and compiles the monitor FMU
Execute	Produces and stores output in <code>data/water_tank_FI_monitor/output</code> directory
Clean	Clears run logs and outputs

RUN THE EXAMPLE

To run the example, change the present directory.

```
1 cd /workspace/examples/digital_twins/water_tank_FI_monitor
```

If required, change the execute permission of lifecycle scripts that need to be executed, for example:

```
1 chmod +x lifecycle/create
```

Now, run the following scripts:

Create

Installs Open Java Development Kit 17 and pip dependencies. The pandas and matplotlib are the pip dependencies installed. The monitor FMU from the NuRV specification is generated and compiled.

```
1 lifecycle/create
```

Execute

Run the co-simulation. Generates the co-simulation output.csv file at `/workspace/examples/data/water_tank_FI_monitor/output`.

```
1 lifecycle/execute
```

Analyze phase

Process the output of co-simulation to produce a plot at: `/workspace/examples/data/water_tank_FI_monitor/output/plots/`.

```
1 lifecycle/analyze
```

A sample plot is given here.

sample plot

In the plot, four colour-coded indicators are used to represent different values. The blue line shows the water tank level, while orange represents the control output and green indicates the monitor's verdict. A red dashed line serves as a reference point, marking the level checked by the monitor. As the water level exceeds this threshold, the monitor's verdict changes from `unknown` (0) to `false` (2).

Examine the results

The results can be found in the `/workspace/examples/data/water_tank_FI_monitor/output` directory.

Run logs can also be viewed in the `/workspace/examples/digital_twins/water_tank_FI_monitor`.

Terminate phase

Clean up the temporary files and delete output plot

```
1 lifecycle/terminate
```

REFERENCES

More details on this case-study can be found in the paper:

```
1 M. Frasheri, C. Thule, H. D. Macedo, K. Lausdahl, P. G. Larsen and
2 L. Esterle, "Fault Injecting Co-simulations for Safety,"
3 2021 5th International Conference on System Reliability and Safety (ICSRS),
4 Palermo, Italy, 2021.
```

The fault-injection plugin is an extension to the Maestro co-orchestration engine that enables injecting inputs and outputs of FMUs in an FMI-based co-simulation with tampered values. More details on the plugin can be found in [fault injection](#) git repository. The source code for this example is also in the same github repository in a [example directory](#).

Incubator Co-Simulation Digital Twin validation with NuRV Monitor

OVERVIEW

This example demonstrates how to validate some digital twin components using FMU monitors (in this example, the monitors are generated with NuRV[1]).

SIMULATED SCENARIO

This example validates some components of the [Incubator digital twin](#), by performing a simulation in which the components are wrapped inside FMUs, and are then inspected at runtime by some a FMU monitor generated by NuRV. Please note that the link to [Incubator digital twin](#) is only provided to know the details of the incubator physical twin. The digital twin (DT) presented here is a co-simulation DT of the Incubator.

The input data for the simulation is generated by a purpose-built FMU component named *source*, which supplies testing data to the *anomaly detector*, simulating an anomaly occurring at time $t=60s$. An additional component, *watcher*, is employed to verify whether the *energy saver* activates in response to an anomaly reported by the *anomaly detector*.

The output of the watcher is the passed to the monitor, which ensures that when an anomaly is detected, the *energy saver* activates within a maximum of three simulation cycles.

EXAMPLE STRUCTURE

A diagram depicting the logical software structure of the example can be seen below.

DT structure

DIGITAL TWIN CONFIGURATION

The example uses the following assets:

Asset Type	Names of Assets	Visibility	Reuse in other Examples
Models	anomaly_detection.fmu	Private	No
	energy_saver.fmu	Private	No
	Source.fmu	Private	No
	Watcher.fmu	Private	No
Specification	safe-operation.smv	Private	No
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes

The *safe-operation.smv* file contains the default monitored specification as described in the [Simulated scenario section](#).

LIFECYCLE PHASES

The lifecycle phases for this example include:

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool Generates and compiles the monitor FMU
Execute	Produces and stores output in data/incubator-NuRV-monitor-validation/output directory
Clean	Clears run logs and outputs

If required, change the execute permissions of lifecycle scripts that need to be executed. This can be done using the following command

```
1 chmod +x lifecycle/{script}
```

where {script} is the name of the script, e.g. *create*, *execute* etc.

RUN THE EXAMPLE

To run the example, change the present directory.

```
1 cd /workspace/examples/digital_twins/incubator-NuRV-monitor-validation
```

If required, change the execute permission of lifecycle scripts that need to be executed, for example:

```
1 chmod +x lifecycle/create
```

Now, run the following scripts:

Create

- Installs Open Java Development Kit 17 in the workspace.
- Generates and compiles the monitor FMU from the NuRV specification

```
1 lifecycle/create
```

Execute

Run the the Digital Twin. Since this is a co-simulation based digital twin, the Maestro co-simulation tool executes co-simulation using the FMU models.

```
1 lifecycle/execute
```

Analyze phase

Process the output of co-simulation to produce a plot at: `/workspace/examples/data/incubator-NuRV-monitor-validation/output/plots`.

```
1 lifecycle/analyze
```

A sample plot is given here.

output plot

In the plot, four colour-coded indicators provide a visual representation of distinct values. The blue line depicts the simulated temperature, while orange one represents the temperature estimate. A red dashed line indicates the target temperature set by the energy saver component. The green line shows the monitor's output verdict. As observed, when there is a disparity between the estimated and actual temperatures, the energy saver adjusts the target temperature downward, indicating that the component is working properly. Thus, the output of the monitor is fixed at `unknown (0)`, signifying that the monitoring property is not violated.

Examine the results

The results can be found in the `/workspace/examples/data/incubator-NuRV-monitor-validation/output` directory where the logs are also included.

Figures of the output results can be found in the `/workspace/examples/data/incubator-NuRV-monitor-validation/output` directory.

Terminate phase

Terminate to clean up the debug files and co-simulation output files.

```
1 lifecycle/terminate
```

REFERENCES

1. More information about NuRV is available on the [NuRV tool page](#).

Incubator Digital Twin with NuRV monitoring service

OVERVIEW

This example demonstrates how a runtime monitoring service (in this example NuRV[1]) can be connected with the [Incubator digital twin](#) to verify runtime behaviour of the Incubator.

SIMULATED SCENARIO

This example simulates a scenario where the lid of the Incubator is removed and later put back on. The Incubator is equipped with anomaly detection capabilities, which can detect anomalous behaviour (i.e. the removal of the lid). When an anomaly is detected, the Incubator triggers an energy saving mode where the heater is turned off.

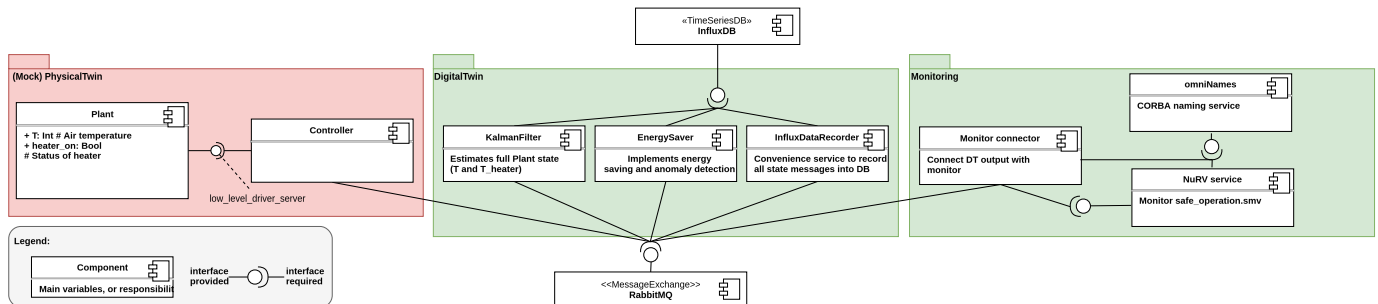
From a monitoring perspective, the aim is to verify that within 3 simulation steps of an anomaly detection, the energy saving mode is turned on. To verify this behaviour, the following property is constructed: . Whenever a True or False verdict is produced by the monitor, it is reset, allowing for the detection of repeated satisfaction/violation detections of the property.

The simulated scenario progresses as follows:

- *Initialization:* The services are initialized and the Kalman filter in the Incubator is given 2 minutes to stabilize. Sometimes, the anomaly detection algorithm will detect an anomaly at startup even though the lid is still on. It will disappear after approx 15 seconds.
- *After 2 minutes:* The lid is lifted and an anomaly is detected. The energy saver is turned on shortly after
- *After another 30 seconds:* The energy saver is manually disabled producing a False verdict.
- *After another 30 seconds:* The lid is put back on and the anomaly detection is given time to detect that the lid is back on. The simulation then ends.

EXAMPLE STRUCTURE

A diagram depicting the logical software structure of the example can be seen below.



The *execute.py* script is responsible for orchestrating and starting all the relevant services in this example. This includes the Incubator DT, CORBA naming service (omniNames) and the NuRV monitor server as well as implementing the *Monitor connector* component that connects the DT output to the NuRV monitor server.

The NuRV monitor server utilizes a CORBA naming service where it registers under a specific name. A user can then query the naming service for the specific name, to obtain a reference to the monitor server. For more information on how the NuRV monitor server works, please refer to [1].

After establishing connection with the NuRV monitor server, the Incubator DT is started and a RabbitMQ client is created that subscribes to changes in the *anomaly* and *energy_saving* states of the DT. Each time an update is received of either state, the full state (the new updated state and the previous other state) is pushed to the NuRV monitor server whereafter the verdict is printed to the console.

DIGITAL TWIN CONFIGURATION

Before running the example, the *simulation.conf* file should be configured with the appropriate RabbitMQ credentials.

The example uses the following assets:

Asset Type	Names of Assets	Visibility	Reuse in other Examples
Service	common/services/NuRV_orbit	Common	Yes
DT	common/digital_twins/incubator	Common	Yes
Specification	safe-operation.smv	Private	No
Script	execute.py	Private	No

The *safe-operation.smv* file contains the default monitored specification as described in the [Simulated scenario section](#). These can be configured as desired.

LIFECYCLE PHASES

The lifecycle phases for this example include:

Lifecycle phase	Completed tasks
create	Downloads the necessary tools and creates a virtual python environment with the necessary dependencies
execute	Runs a python script that starts up the necessary services as well as the Incubator simulation. Various status messages are printed to the console, including the monitored system states and monitor verdict.
clean	Removes created <i>data</i> directory and incubator log files.

If required, change the execute permissions of lifecycle scripts that need to be executed. This can be done using the following command

```
1 chmod +x lifecycle/{script}
```

where {script} is the name of the script, e.g. *create*, *execute* etc.

RUNNING THE EXAMPLE

To run the example, first run the following command in a terminal:

```
1 cd /workspace/examples/digital_twins/incubator-monitor-server/
```

Then, first execute the *create* script (this can take a few minutes depending on the network connection) followed by the *execute* script using the following command:

```
1 lifecycle/{script}
```

The *execute* script will then start outputting system states and the monitor verdict approx every 3 seconds. The output is printed as follows **"State: {anomaly state} & {energy_saving state}, verdict: {Verdict}"** where "*anomaly*" indicates that an anomaly is detected and "*!anomaly*" indicates that an anomaly is not currently detected. The same format is used for the *energy_saving* state.

The monitor verdict can be True, False or Unknown, where the latter indicates that the monitor does not yet have sufficient information to determine the satisfaction of the property.

An example output trace is provided below:

```
1  ....
2  Running scenario with initial state: lid closed and energy saver on
3  Setting energy saver mode: enable
4  Setting G_box to: 0.5763498
5  State: !anomaly & !energy_saving, verdict: True
6  State: !anomaly & !energy_saving, verdict: True
7  ....
8  State: anomaly & !energy_saving, verdict: Unknown
9  State: anomaly & energy_saving, verdict: True
10 State: anomaly & energy_saving, verdict: True
```

There is currently some startup issues with connecting to the NuRV server, and it will likely take a few tries before the connection is established. This is however handled automatically.

REFERENCES

1. Information on the NuRV monitor can be found on [FBK website](#).

Incubator Digital Twin with NuRV FMU Monitoring Service

OVERVIEW

This example demonstrates how an FMU can be used as a runtime monitoring service (in this example NuRV[1]) and connected with the [Incubator digital twin](#) to verify runtime behaviour of the Incubator.

SIMULATED SCENARIO

This example simulates a scenario where the lid of the Incubator is removed and later put back on. The Incubator is equipped with anomaly detection capabilities, which can detect anomalous behaviour (i.e. the removal of the lid). When an anomaly is detected, the Incubator triggers an energy saving mode where the heater is turned off.

From a monitoring perspective, the aim is to verify that within 3 messages of an anomaly detection, the energy saving mode is turned on. To verify this behaviour, the following property is constructed:

.

The monitor will output the *unknown* state as long as the property is satisfied and will transition to the *false* state once a violation is detected.

The simulated scenario progresses as follows:

- *Initialization*: The services are initialized and the Kalman filter in the Incubator is given 2 minutes to stabilize. Sometimes, the anomaly detection algorithm will detect an anomaly at startup even though the lid is still on. It will disappear after approx 15 seconds.
- *After 2 minutes*: The lid is lifted and an anomaly is detected. The energy saver is turned on shortly after.
- *After another 30 seconds*: The energy saver is manually disabled producing a *false* verdict.
- *After another 30 seconds*: The lid is put back on and the anomaly detection is given time to detect that the lid is back on. The monitor is then reset producing an Unknown verdict again. The simulation then ends.

EXAMPLE STRUCTURE

A diagram depicting the logical software structure of the example can be seen below.

DT structure

The *execute* script is responsible for starting the NuRV service and running the Python script that controls the scenario (*execute.py*).

The *execute.py* script starts the Incubator services and runs the example scenario. Once the Incubator DT is started, a RabbitMQ client is created that subscribes to changes in the *anomaly* and *energy_saving* states of the DT, as well as the verdicts produced by the NuRV service. Each time an update is received, the full state and verdict is printed to the console.

DIGITAL TWIN CONFIGURATION

Before running the example, the *simulation.conf* file should be configured with the appropriate RabbitMQ credentials.

The example uses the following assets:

Asset Type	Names of Assets	Visibility	Reuse in other Examples
Tools	common/tool/NuRV/NuRV	Common	Yes
Other	common/fmi2_headers	Common	Yes
DT	common/digital_twins/incubator	Common	Yes
Specification	safe-operation.smv	Private	No
Script	execute.py	Private	No

The *safe-operation.smv* file contains the default monitored specification as described in the [Simulated scenario section](#). These can be configured as desired.

LIFECYCLE PHASES

The lifecycle phases for this example include:

Lifecycle phase	Completed tasks
create	Downloads the necessary tools and creates a virtual python environment with the necessary dependencies
execute	Runs a python script that starts up the necessary services as well as the Incubator simulation. Various status messages are printed to the console, including the monitored system states and monitor verdict.

If required, change the execute permissions of lifecycle scripts that need to be executed. This can be done using the following command.

```
1 chmod +x lifecycle/{script}
```

where {script} is the name of the script, e.g. *create*, *execute* etc.

RUNNING THE EXAMPLE

To run the example, first run the following command in a terminal:

```
1 cd /workspace/examples/digital_twins/incubator-NuRV-fmu-monitor-service/
```

Then, first execute the *create* script followed by the *execute* script using the following command:

```
1 lifecycle/{script}
```

The *execute* script will then start outputting system states and the monitor verdict approx every 3 seconds. The output is printed as follows.

"State: {anomaly state} & {energy_saving state}"

where "*anomaly*" indicates that an anomaly is detected and "*!anomaly*" indicates that an anomaly is not currently detected. The same format is used for the *energy_saving* state. NuRV verdicts are printed as follows

"Verdict from NuRV: {verdict}".

The monitor verdict can be false or unknown, where the latter indicates that the monitor does not yet have sufficient information to determine the satisfaction of the property. The monitor will never produce a true verdict as the entire trace must be verified to ensure satisfaction due to the G operator. Thus the unknown state can be viewed as a tentative true verdict.

An example output trace is provided below:

```

1  ....
2  Using LIFECYCLE_PATH: /workspace/examples/digital_twins/incubator-NuRV-fmu-monitor-service/lifecycle
3  Using INCUBATOR_PATH: /workspace/examples/digital_twins/incubator-NuRV-fmu-monitor-service/lifecycle/../../../../common/digital_twins/incubator
4  Starting NuRV FMU Monitor Service, see output at /tmp/nurv-fmu-service.log
5  NuRVService.py PID: 13496
6  Starting incubator
7  Connected to rabbitmq server.
8  Running scenario with initial state: lid closed and energy saver on
9  Setting energy saver mode: enable
10 Setting G_box to: 0.5763498
11 State: !anomaly & !energy_saving
12 State: !anomaly & !energy_saving
13 Verdict from NuRV: unknown
14 State: !anomaly & !energy_saving
15 State: !anomaly & !energy_saving
16 Verdict from NuRV: unknown
17 State: !anomaly & !energy_saving
18 State: !anomaly & !energy_saving
19 Verdict from NuRV: unknown
20 State: !anomaly & !energy_saving
21 State: !anomaly & !energy_saving
22 Verdict from NuRV: unknown
23 State: !anomaly & !energy_saving
24 State: !anomaly & !energy_saving
25 Verdict from NuRV: unknown

```

REFERENCES

1. Information on the NuRV monitor can be found on [FBK website](#).

3.8.3 Automation

Automation Examples

These examples demonstrate the use of DevOps features in the DTaaS platform. GitLab CI/CD pipelines automate the lifecycle management of digital twins, enabling create, execute, and clean phases to run without manual intervention.

AVAILABLE EXAMPLES

Example	Description
Mass Spring Damper	Co-simulation DT with automated lifecycle via GitLab CI/CD
Structural Health Monitoring	SHM DT with automated system identification and model update
Heart Digital Twin	Real-time cardiovascular monitoring with web interface
Heart Digital Twin (PaaS)	Cloud deployment of Heart DT using PaaS strategies

CODEBASE

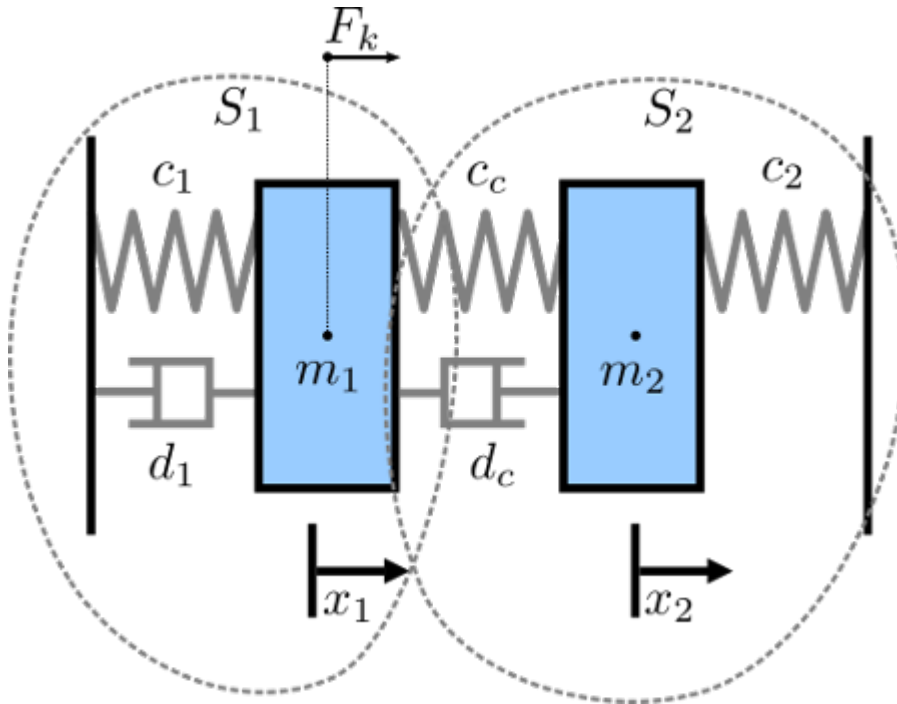
The source code for these examples is available in the [DTaaS-examples repository](#).

Mass Spring Damper

OVERVIEW

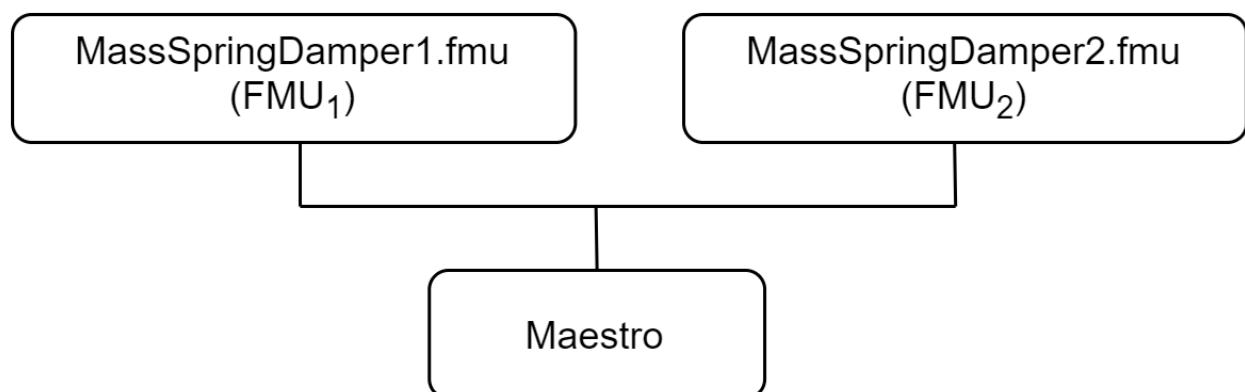
The mass spring damper digital twin (DT) comprises two mass spring dampers and demonstrates how a co-simulation based DT can be used within DTaaS.

EXAMPLE DIAGRAM



EXAMPLE STRUCTURE

There are two simulators included in the study, each representing a mass spring damper system. The first simulator calculates the mass displacement and speed of for a given force acting on mass . The second simulator calculates force given a displacement and speed of mass . By coupling these simulators, the evolution of the position of the two masses is computed.



DIGITAL TWIN CONFIGURATION

This example uses two models and one tool. The specific assets used are:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Models	MassSpringDamper1.fmu	Private	Yes
	MassSpringDamper2.fmu	Private	Yes
Tool	maestro-2.3.0-jar-with-dependencies.jar	Common	Yes

The `co-sim.json` and `time.json` are two DT configuration files used for executing the digital twin. These two files can be modified to customise the DT as required.

LIFECYCLE PHASES

Lifecycle Phase	Completed Tasks
Create	Installs Java Development Kit for Maestro tool
Execute	Produces and stores output in data/mass-spring-damper/output directory
Clean	Clears run logs and outputs

DEVOPS AUTOMATION

This example demonstrates the use of DevOps features of the DTaaS platform. The GitLab DevOps pipelines are used for providing this feature in the DTaaS.

The `.gitlab-ci.yml` file controls the sequence of executing the lifecycle scripts of the example. The configuration format of `.gitlab-ci.yml` permits specifying stages for execution of a program. In this first example, all the lifecycle scripts are put in single stage, namely *build_and_run*.

This example produces co-simulation outputs which are then saved in the artefacts repository of the GitLab. These can be accessed at <https://<gitlab-host>/<group>/<username>/-/artifacts>. Replace `<gitlab-host>`, `<group>`, and `<username>` with the values for your GitLab instance and project.

RUN THE EXAMPLE

To run the example, navigate to the following directory.

```
1 cd /workspace/examples/digital_twins/mass-spring-damper
```

If required, change the execute permission of the lifecycle scripts to be executed, for example:

```
1 chmod +x lifecycle/create
```

Now, run the following scripts:

Create

Installs Open Java Development Kit 17 in the workspace.

```
1 lifecycle/create
```

Execute

Run the Digital Twin. Since this is a co-simulation based digital twin, the Maestro co-simulation tool executes co-simulation using the two FMU models.

```
1 lifecycle/execute
```

Examine the results

The results can be found in the `/workspace/examples/data/mass-spring-damper/output` directory.

You can also view run logs in the `/workspace/examples/digital_twins/mass-spring-damper`.

Terminate phase

Terminate to clean up the debug files and co-simulation output files.

```
1 lifecycle/terminate
```

REFERENCES

More information about co-simulation techniques and mass spring damper case study are available in:

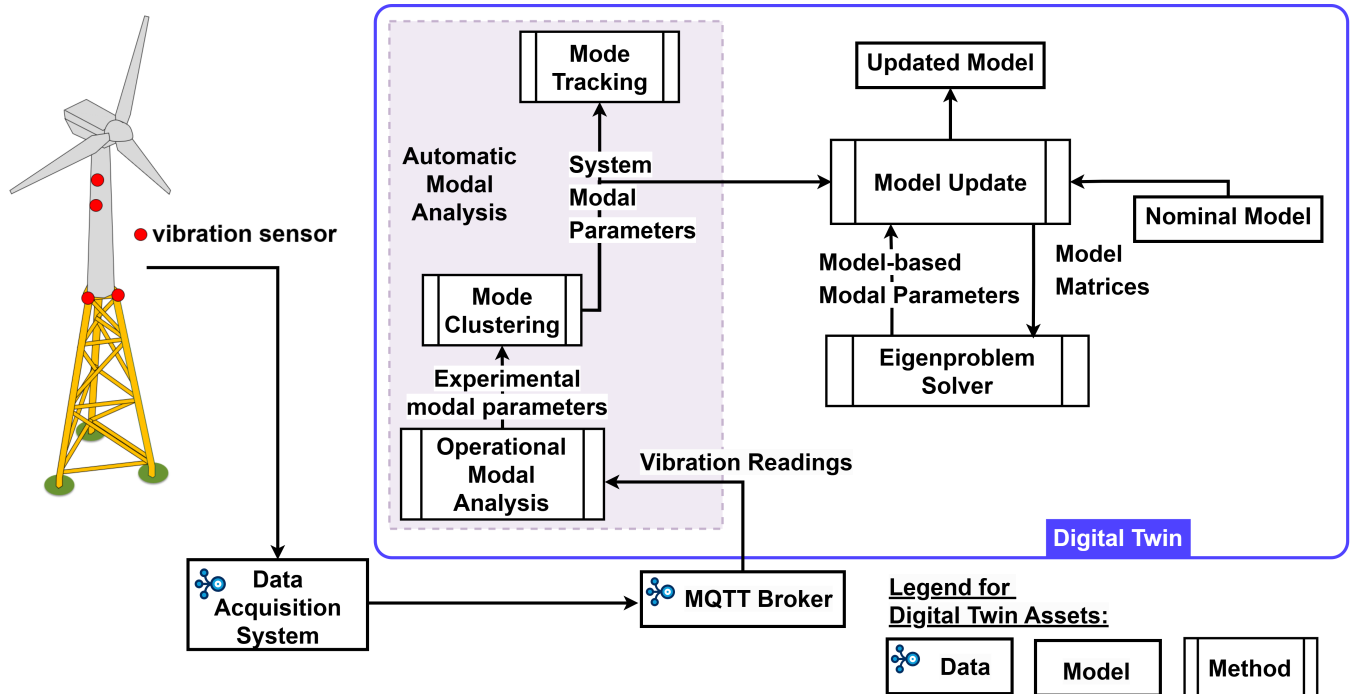
```
1 Gomes, Cláudio, et al. "Co-simulation: State of the art."  
2 arXiv preprint arXiv:1702.00686 (2017).
```

The source code for the models used in this DT are available in [mass spring damper](#) github repository.

Structural Health Monitoring

OVERVIEW

This example demonstrates the use of digital twin methodology for structural health monitoring (SHM). A digital twin workflow shown below has been developed for the SHM use cases.



The complete source code for this example is [available online](#).

EXAMPLE STRUCTURE

This digital twin consists of

- **SHM package:** package implementing the digital twin workflow. This package also contains the recorded data from physical twin and replay functions that stream data into the digital twin workflow.
- **replay.json:** MQTT configuration file for replaying the recorded data

The package can demonstrate these experimental scenarios.

- **replay** replays the recorded readings from cantilever beam setup.
- **acceleration_readings** demonstrates the use of `Accelerometer` class to extract accelerometer measurements from MQTT data stream.
- **aligning_readings** demonstrates the use of `Aligner` class to collect and align accelerometer measurements from multiple MQTT data streams.
- **sysid** demonstrates the use of `sysid` with four cases:
 - a. **sysid-and-plot**: plots natural frequencies.
 - b. **sysid-and-print**: prints sysid output to console.
 - c. **sysid-and-publish**: publishes one set of sysid output via MQTT to the config given under [sysid] config.
 - d. **live-sysid-and-publish**: Continuously publishes sysid output via MQTT to the config given under [sysid] config.
- **Clustering** demonstrates the use of `clustering` with three cases:
 - a. **clustering-with-local-sysid**: gets the sysid output by running sysid locally, then runs the mode clustering.
 - b. **clustering-with-remote-sysid**: gets sysid output by subscribing, then runs the mode clustering. This is a one time operation.
 - c. **live-clustering-with-remote-sysid**: gets sysid output by subscribing, then runs the mode clustering. This operation runs in loop.
 - d. **live-clustering-with-remote-sysid-and-publish**: gets sysid output by subscribing, then runs the mode clustering. The cluster results are published. This operation runs in loop.
- **mode-tracking** demonstrates the use of `mode_tracking` with three cases:
 - a. **mode-tracking-with-local-sysid**: gets the sysid output by running sysid locally, then runs mode clustering and mode tracking.
 - b. **mode-tracking-with-remote-sysid**: gets sysid output by subscribing, then runs mode clustering and mode tracking. This is a one time operation.
 - c. **live-mode-tracking-with-remote-sysid**: gets sysid output by subscribing, then runs mode clustering and mode tracking. This operation runs in loop.
- **model-update** demonstrates the use of `model_update` with two cases:
 - a. **model-update-local-sysid**: gets the sysid output, then uses it to run update model and get updated system parameters.
 - b. **live-model-update-with-remote-sysid**: gets the sysid output by subscribing to MQTT topic, then runs mode clustering to run update model and get updated system parameters.
 - c. **live-model-update-with-remote-clustering**: gets the mode clustering output by subscribing to MQTT topic, then uses the mode clustering output to run update model and get updated system parameters.

Of the above mentioned experimental scenarios, the following can be placed in the automated execution mode.

- **acceleration_readings** demonstrates the use of `Accelerometer` class to extract accelerometer measurements from MQTT data stream.
- **aligning_readings** demonstrates the use of `Aligner` class to collect and align accelerometer measurements from multiple MQTT data streams.
- **sysid** demonstrates the use of `sysid` with four cases. It prints sysid output to console.
- **Clustering** demonstrates the use of `sysid` and `clustering`.
- **mode-tracking** demonstrates the use of `sysid` and `mode_tracking`.
- **model-update** demonstrates the use of the digital twin workflow upto `model_update`.

CONFIGURATION

This example uses MQTT broker for replaying the recorded data from physical twin. The format of the configuration file is in `replay.json`. MQTT credentials need to be updated in the file. In case a local MQTT broker is not available, [test server](#) can also be used.

USE

The available experimental scenarios can be found by running the program

```

1  $example-shm
2  Usage: example-shm [OPTIONS] COMMAND [ARGS]...
3
4  Options:
5    --config TEXT  Path to config file
6    --help         Show this message and exit.
7
8  Commands:
9    accelerometers
10   align-readings
11   clustering-with-local-sysid
12   clustering-with-remote-sysid
13   live-clustering-with-remote-sysid
14   live-clustering-with-remote-sysid-and-publish
15   live-mode-tracking-with-remote-sysid
16   live-model-update-with-remote-clustering
17   live-model-update-with-remote-sysid
18   live-sysid-publish
19   mode-tracking-with-local-sysid
20   mode-tracking-with-remote-sysid
21   model-update-with-local-sysid
22   replay
23   sysid-and-plot
24   sysid-and-print
25   sysid-and-publish

```

Run these examples with the `replay.json` config:

```

1  $example-shm --config replay.json <experiment-name>
2  Usage: example-shm --config replay.json accelerometers

```

Automated Execution

The current configuration of this example demonstrates system identification and model update. A truncated sample output can be seen here. The execution timestamps have been removed from this log to show the main content.

```

1  Using docker image sha256:16f554efe9f9e182e2ec6b86ad37360138407a456a69982ed1b8261716ce5163 for python:3.12 with digest
2  $ cd digital_twins/shm
3  $ bash lifecycle/sysid_print
4  ...
5
6  pyoma2.setup.base - INFO - Running SSICovmm_mt... (base:123)
7  pyoma2.functions.ssi - INFO - Assembling Hankel matrix method: cov_mm... (ssi:82)
8  pyoma2.functions.ssi - INFO - ... uncertainty calculations... (ssi:94)
9  pyoma2.functions.ssi - INFO - SSI for increasing model order... (ssi:359)
10 JSON configuration loaded successfully.
11 on_connect: Connected with response code Success
12 Subscribing to topic: cpsens/recorded/1/metadata
13 on_subscribe: Subscription ID 1 with QoS levels [ReasonCode(Suback, 'Granted QoS 0')]
14 Extracted Fs from metadata: 256.0
15 on_connect: Connected with response code Success
16 Subscribing to topic: cpsens/recorded/1/data
17 on_subscribe: Subscription ID 1 with QoS levels [ReasonCode(Suback, 'Granted QoS 1')]
18 on_subscribe: Subscription ID 2 with QoS levels [ReasonCode(Suback, 'Granted QoS 1')]
19 on_subscribe: Subscription ID 3 with QoS levels [ReasonCode(Suback, 'Granted QoS 1')]
20 on_subscribe: Subscription ID 4 with QoS levels [ReasonCode(Suback, 'Granted QoS 1')]
21 on_subscribe: Subscription ID 5 with QoS levels [ReasonCode(Suback, 'Granted QoS 1')]
22 on_subscribe: Subscription ID 6 with QoS levels [ReasonCode(Suback, 'Granted QoS 1')]
23 on_subscribe: Subscription ID 7 with QoS levels [ReasonCode(Suback, 'Granted QoS 1')]
24 on_subscribe: Subscription ID 8 with QoS levels [ReasonCode(Suback, 'Granted QoS 1')]
25 Waiting for data for 0.1 seconds
26 Waiting for data for 0.2 seconds
27 Waiting for data for 0.3 seconds
28 ...
29
30 sysid parameters: {'freq_variance_threshold': 0.1, 'damp_variance_threshold': 1000000, 'Fs': 256, 'model_order_min': 2, 'model_order': 15, 'block_shift':
31 30, 'mstab': 6, 'tMAC': 0.95, 'bound_multiplier': 2, 'alignment_factor': [0.05, 0.01], 'phi_cri': 0.8, 'freq_cri': 0.2, 'obj_cri': 0.1, 'tMAC_MU': 0.7,
32 'modes_search_paring': 6, 'pars_to_update': ['k_rot', 'm'], 'MU_start_values': array([10. , 0.015]), 'MU_bounds': [(0.01, 1000), (0, 1000)],
33 'MU_modes': [1, 2, 3]}
34
35 0%|          | 0/16 [00:00<?, ?it/s]
36 100%|██████████| 16/16 [00:00<00:00, 23563.51it/s]
37 Cluster 0.4465797245502472 too short: 1 Must be: > 6
38 Cluster saved: 27.93849468231201
39 Updated parameters are:
40 k_rot: 12.04591179878987
41 m: 0.0
Model frequencies: [ 4.13819299 26.68572155 80.22633861 164.67016633 180.37962416
326.70816499] [Hz]
[DONE].

```

REFERENCES

1. This example corresponds to [commit ce47244](#) of the [example-shm repository](#).
2. Talasila, P, Tcherniak, D., Jensen, A. M. D., Mahato, S., Schörghofer-Queiroz, A., Ulriksen, M. D., ... & Damkilde, L. (2025, July). Structural Health Monitoring of Engineering Structures Using Digital Twins: A Digital Twin Platform Approach. In International Conference on Experimental Vibration Analysis for Civil Engineering Structures (pp. 986-996). Cham: Springer Nature Switzerland.)

Heart Digital Twin

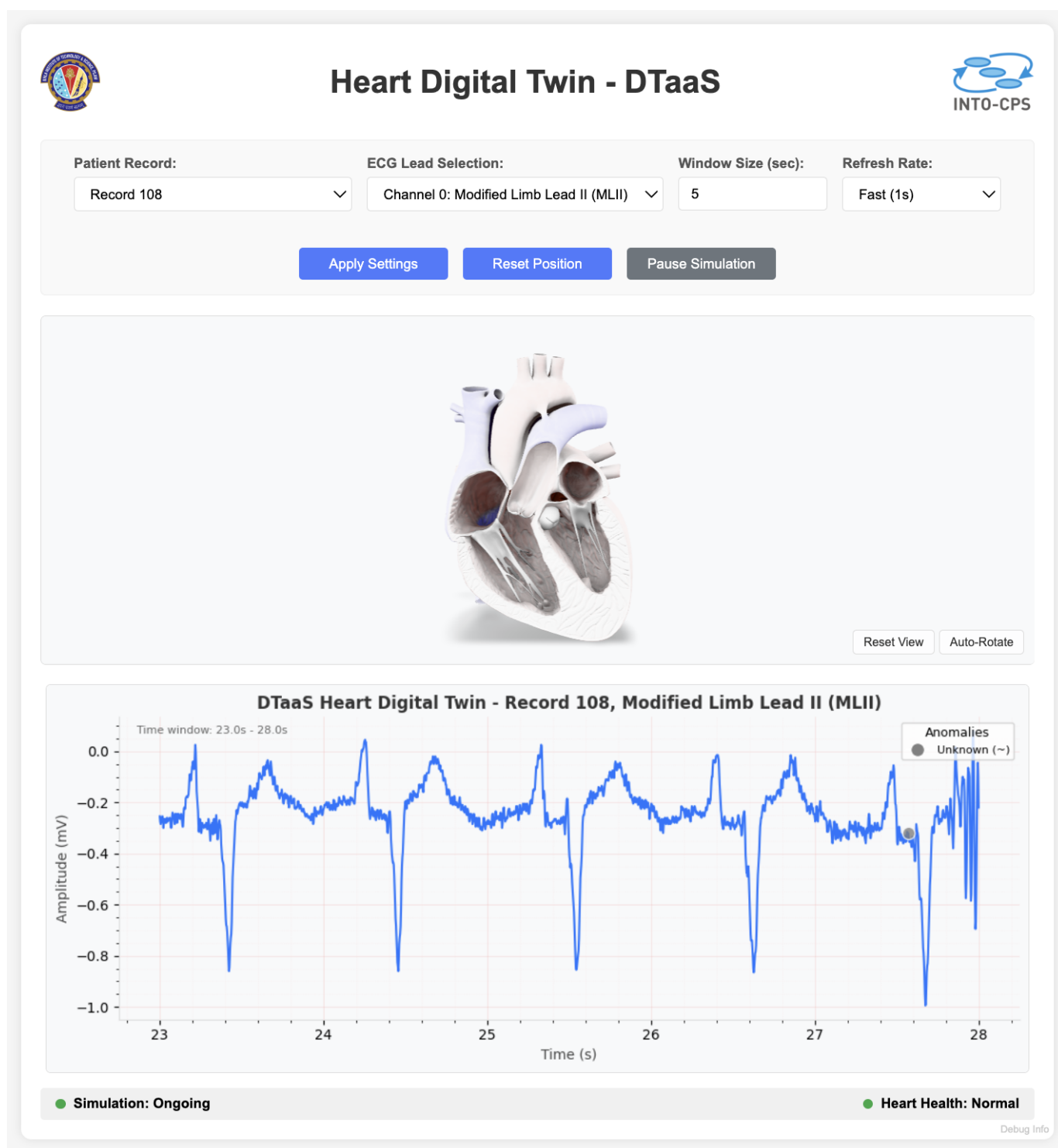
OVERVIEW

The Heart Digital Twin demonstrates a real-time cardiovascular monitoring system using ECG data from the MIT-BIH Arrhythmia Database. This digital twin provides live ECG visualisation with anomaly detection and features an interactive 3D heart model that synchronises with the cardiac rhythm, creating an immersive healthcare monitoring experience within the DTaaS platform.

EXAMPLE STRUCTURE

The Heart Digital Twin consists of several key components working together:

- **ECG Data Processing:** Real-time processing and visualisation of electrocardiogram signals from the MIT-BIH database
- **3D Heart visualisation:** Interactive 3D heart model with synchronised pumping animation
- **Anomaly Detection:** Real-time identification and highlighting of cardiac anomalies
- **Web Interface:** Flask-based web application for real-time monitoring and control



MIT-BIH ARRHYTHMIA DATABASE

This digital twin utilises the renowned **MIT-BIH Arrhythmia Database**, which contains 48 half-hour excerpts of two-channel ambulatory ECG recordings. The database includes:

- Records from 47 subjects studied by the BIH Arrhythmia Laboratory
- 360 Hz sampling frequency with 11-bit resolution
- Expert annotations for arrhythmia detection and classification
- Multiple ECG channels (MLII, V1, V2, V4, V5) depending on the record

Records 100-109 are available for simulation, each containing different types of cardiac rhythms and anomalies for comprehensive testing and demonstration.

DIGITAL TWIN CONFIGURATION

This example uses the following assets:

Asset Type	Names of Assets	Visibility	Reuse in Other Examples
Data	MIT-BIH Arrhythmia Database Records (100-109)	Public	Yes
Models	3D Heart Model (Beating heart.glb)	Public	Yes
Functions	ECG Processing & Anomaly Detection	Public	Yes
Tools	Flask Web Server & UI Components	Public	Yes

The heart digital twin can be customised through various parameters:

- **Record Selection:** Choose from records 100-109
- **Channel Selection:** Switch between available ECG channels
- **Window Size:** Adjust the time window for real-time display
- **Update Frequency:** Control the refresh rate of the simulation

3D HEART ANIMATION

The digital twin features a **synchronised 3D heart pumping animation** that enhances the monitoring experience:

- **Real-time Synchronisation:** Heart model animation matches the ECG rhythm
- **Interactive Controls:**
 - Camera controls for 360° viewing
 - Auto-rotate functionality
 - Reset view option
- **Animation Features:**
 - Realistic beating heart model
 - Smooth cardiac cycle animation
 - Pause/resume functionality synchronised with ECG simulation
- **Model Format:** High-quality GLB (GL Transmission Format Binary) 3D model
- **Rendering:** WebGL-based rendering using model-viewer component

The 3D heart provides visual feedback that correlates with the electrical activity shown in the ECG, making it easier to understand the relationship between electrical signals and mechanical heart function.

LIFECYCLE PHASES

Lifecycle Phase	Completed Tasks
Create	Sets up Python virtual environment and installs required dependencies (NumPy, Pandas, Matplotlib, WFDB, Flask)
Execute	Starts the Flask web server and launches the heart monitoring interface accessible via web browser
Clean	Terminates the running application and removes the virtual environment

DEVOPS AUTOMATION

This example demonstrates the use of DevOps features of the DTaaS platform. GitLab DevOps pipelines are used to automate and manage its lifecycle.

Pipeline Configuration

The `.gitlab-ci.yml` file controls the sequence of lifecycle script executions for the heart digital twin. It defines multiple stages:

- **create:** Sets up the Python virtual environment and installs dependencies
- **execute:** Runs the heart digital twin application

Runner Requirements

Important: This digital twin requires a **shell-based GitLab runner**, not a Docker-based runner. The application needs:

- Access to the host system's Python environment
- Ability to create and manage Python virtual environments
- File system access for model loading and data processing
- Network access for the web application

Docker-based runners may restrict network or filesystem access critical to this digital twin.

Alternative Execution Mode: Docker

There is also an option to use **Docker-based execution** for the heart digital twin — while still using a **shell-based GitLab runner**.

The corresponding GitLab CI file for this approach is `gitlab-ci-docker.yml`

To enable this alternative execution path, the root `.gitlab-ci.yml` must be modified to reference this Docker-based pipeline (`gitlab-ci-docker.yml`) instead of the default one.

Artifacts and Outputs

This example produces:

- Heart simulation outputs
- Web application logs

These are saved as artefacts in the GitLab pipeline. The virtual environment created in the `create` stage is also preserved as an artefact for reuse in the `execute` stage.

RUN THE EXAMPLE

To run the example, navigate to the following directory:

```
1 cd /workspace/examples/digital_twins/heart_dt
```

If required, change the execute permission of the lifecycle scripts to be executed:

```
1 chmod +x lifecycle/create
2 chmod +x lifecycle/execute
3 chmod +x lifecycle/clean
```

Now, run the following scripts:

Create

Sets up the Python environment and installs all required dependencies including WFDB for MIT-BIH data access, Flask for the web interface, and scientific computing libraries.

```
1 lifecycle/create
```

Execute

Launches the Heart Digital Twin web application. The server will start and be accessible via web browser for real-time ECG monitoring and 3D heart visualisation.

```
1 lifecycle/execute
```

The application will be available at `http://localhost:5001` and will automatically:

- Load required MIT-BIH records
- Initialise the ECG data processing
- Start real-time visualisation
- Enable 3D heart model Synchronisation

Examine the results

Access the web interface to:

- **Monitor Real-time ECG:** View live electrocardiogram signals
- **Control Simulation:** Select different records, channels, and update frequencies
- **Interact with 3D Heart:** Rotate, zoom, and control the heart animation
- **Detect Anomalies:** Observe highlighted cardiac irregularities
- **Pause/Resume:** Control the simulation flow

Application logs can be found in the `logs/heart_dt.log` file for debugging and monitoring purposes.

Clean

Terminates the running application and cleans up the environment.

```
1 lifecycle/clean
```

This will:

- Stop the Flask web server
- Remove the Python virtual environment
- Clean up any temporary files

REFERENCES

More information about the MIT-BIH Arrhythmia Database and cardiac monitoring:

```
1 Moody GB, Mark RG. The impact of the MIT-BIH Arrhythmia Database.
2 IEEE Eng in Med and Biol 20(3):45-50 (May-June 2001).
```

```
1 Goldberger, A., et al. PhysioBank, PhysioToolkit, and PhysioNet:
2 Components of a new research resource for complex physiologic signals.
3 Circulation 101(23):e215-e220, 2000.
```

The MIT-BIH Arrhythmia Database is available at: <https://physionet.org/content/mitdb/>

Heart Digital Twin - PaaS Edition

OVERVIEW

This example demonstrates the use of DevOps features for the Heart Digital Twin in a **Platform-as-a-Service (PaaS)** deployment mode. This version is specifically designed for cloud deployment and supports multiple deployment strategies including Render.com and GitLab CI/CD with GitHub mirroring.

DEVOPS AUTOMATION

Setup Instructions

1. **Connect GitHub Repository to Render.com:**
2. Sign up for a Render.com account
3. Connect the GitHub repository
4. **Automatic Deployment:**
5. Render.com automatically deploys on detecting changes in the GitHub repository

The `.gitlab-ci.yml` file provides a mirror-to-GitHub pipeline that:

- Automatically syncs changes from GitLab to a GitHub repository
- Enables GitHub-based deployments (like Render.com) to stay updated
- Requires GitLab CI/CD variables: `GITHUB_USERNAME` and `GITHUB_TOKEN`

REFERENCES

See the [Heart Digital Twin](#) documentation for details about the underlying cardiovascular monitoring functionality.