BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

GRAPHICS EDITOR FOR COMPUTATIONAL WORKFLOWS IN TOSCA FORMAT

GRAFICKÝ EDITOR VÝPOČETNÍCH PROCESŮ VE FORMÁTU TOSCA

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **JAN SWIATKOWSKI**
AUTOR PRÁCE

SUPERVISOR **doc. Ing. JIŘÍ JAROŠ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

147291

Institut:	Department of Computer Systems (UPSY)
Student:	**Swiatkowski Jan**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Graphics Editor for Computational Workflows in TOSCA Format**
Category:	User Interfaces
Academic year:	2022/23

Assignment:

1. Familiarize yourself with tools and use cases of scientific computational workflows.
2. Review data types and algorithms for construction of scientific workflows.
3. Design a simple editor for construction of scientific workflows generating TOSCA prescriptions for Alien4Cloud.
4. Implement the designed solution so that it allows to construct a simple workflow containing one HPC or cloud job with inputs and outputs.
5. Evaluate implemented solution, write up a user manual and discuss possible future extensions.

Literature:

- According to supervisor's advice.

Requirements for the semestral defence:

- Items 1 to 3 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Jaroš Jiří, doc. Ing., Ph.D.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	4.5.2023

Abstract

This thesis presents the design and implementation of a graphical workflow editor aimed at non-computer scientists, which enables the creation of complex computational workflows with minimal technical knowledge. The editor provides a graphical interface for composing workflow from computational tasks with supports a variety of input and output types, including value-based and dataset-based inputs and outputs. The resulting workflows can be exported to the TOSCA workflow specification, making them compatible with the LEXIS platform. The editor was developed in .NET Blazor Server framework in C# and JavaScript and employs the JointJS library for creating the graphical representation of workflows. The resulting tool provides an accessible means for researchers and other non-technical users to compose and execute advanced computational workflows.

Abstrakt

Tato práce se zabývá návrhem a implementací grafického editoru pracovních toků zaměřeného na vědce, kteří nejsou z oboru informačních technologií, a editor jim umožňuje vytváření složitých výpočetních toků s minimální technickou znalostí. Editor poskytuje grafické rozhraní pro sestavení toků z výpočetních úloh s podporou různých typů vstupů a výstupů, včetně hodnotových a datových vstupů a výstupů. Výsledné toky lze exportovat do specifikace TOSCA popisující pracovní tok, což umožňuje jejich použití na platformě LEXIS. Editor byl vyvinut v rámci .NET Blazor Server frameworku v jazyce C# a JavaScript a využívá knihovnu JointJS pro vytváření grafické reprezentace toků. Výsledný nástroj poskytuje dostupný způsob, jak vytvářet a spouštět pokročilé výpočetní toky pro výzkumníky a další uživatele.

Keywords

OASIS TOSCA, Computational Workflow, Scientific Workflow, Graphical Editor, HPC, Cloud, C#, JavaScript, .NET Blazor Server

Klíčová slova

OASIS TOSCA, výpočetní pracovní tok, grafický editor, HPC, Cloud, C#, JavaScript, .NET Blazor Server

Reference

SWIATKOWSKI, Jan. *Graphics Editor for Computational Workflows in TOSCA Format*. Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Jiří Jaroš, Ph.D.

Graphics Editor for Computational Workflows in TOSCA Format

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of doc. Ing. Jiří Jaroš, Ph.D.
The supplementary information was provided by Ing. Kateřina Slaninová, Ph.D.,
Ing. Jan Martinovič, Ph.D., Ing. Martin Golasowski, Ph.D.
and doc. Mgr. Jiří Dvorský, Ph.D.
I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Jan Swiatkowski
May 10, 2023

Acknowledgements

I would like to express my heartfelt gratitude to the entire LEXIS platform team at IT4Innovations for their invaluable guidance and support throughout the writing of this thesis and the development of the workflow editor.
I am particularly grateful to Ing. Kateřina Slaninová, Ph.D., Ing. Jan Martinovič, Ph.D., Ing. Martin Golasowski, Ph.D. and my supervisor doc. Ing. Jiří Jaroš, Ph.D. for their valuable insights and encouragement.

Contents

List of Figures

Chapter 1

Introduction

In today's world, even the most qualified scientific experts may require additional knowledge to effectively use traditional High-Performance Computing (HPC)[24] environments for their research. Additionally, HPC environment users are rarely familiar with Linux operation systems, shell terminal or orchestration tools. Hence, the users are not necessarily experienced with managing their computational applications directly on high-performance clusters.

The support of non-computer scientists' need to compute non-trivial solutions on high-performance computing clusters was the motivation behind the creation of a computational workflow[46] editor. The computational workflows may be a constituent of the scientific workflow. „A scientific workflow is the description of a process for accomplishing a scientific objective, usually expressed in terms of tasks and their dependencies. Typically, scientific workflow tasks are computational steps for scientific simulations or data analysis steps.“[41] The article „Characterization of Scientific Workflows“[30] demonstrates some basic workflow structures in Fig. 1. These abstract structures are similar to those used in computational workflows, which consist of series of computational tasks. The goal is to create a combination of high-performance computing (HPC) and cloud tasks that can be run in parallel or as a pipeline, along with data preprocessing, postprocessing, aggregation, and distribution. The goal of the editor is to empower non-computer scientists to create computational workflows with the help of workflow architects. To achieve this goal, the editor leverages the LEXIS[15] system, which provides HPC-as-a-Service and is intended to complement or enhance LEXIS. Based on my experience, most scientist compute on their small clusters or personal computers. The HPC cluster offers more computational power and options for their computations, although the usage complexity proves to be an issue. The editor aims to help users create and compose workflows from straightforward elements. The basic expected operations are:

- Select computational applications
- Enter the computational parameters or data inputs
- Connect the data inputs and outputs between the computational applications in case of application chaining
- Fetch the source data into the application and save the output data

More advanced requirements could be to run more than one instance of the application in parallel to create a race. For example, the race case may be beneficial in urgent computing

like *Fast Tsunami Simulations for a Real-Time Emergency Response Flow* [37]. Further possible scenario is running computational applications instances based on the list of input datasets, i.e. one instance of a computational application will be orchestrated for each dataset in the list.

1.1 Thesis objectives

The objective of this thesis is to analyze and evaluate alternative tools available on the market for creating scientific and computational workflows. The work investigates abstract structures and algorithms used within the given challenges, and describes in detail the design and implementation of a minimal editor that enables users to compose basic computational workflows. The end product of the thesis is a workflow editor that supports export to the TOSCA workflow specification for the LEXIS platform, aimed at facilitating the creation of scientific workflows by non-computer scientists.

1.2 Use case

1.2.1 Use case: Avio Aero Turbomachinery

A real-world example of a potential use case of the editor could be the turbomachinery computational workflow described in LEXIS deliverable 5.5 [42]. HPC computational job utilizes a CFD solver nanoFluidX for simulating behaviour of an air-oil frozen field mixture in a gearbox. The computational workflow steps are preprocessing, simulation runtime, and postprocessing.

1.2.2 Use case: Loschmidt Laboratory Tools

Loschmidt Laboratory[1] offers plenty of tools for biochemical scientists, mainly protein-focused. As the end-users of their tools are mostly scientists with basic knowledge of computer science, the end-users are more friendly with the graphical interface of their tools. However, some of the computational parts of the tools may require high-performance machines and may be helpful to compute them on HPC clusters on demand. For example, on the diagram of the Fireprot-ASR tool [4], we can see a more complicated computational workflow than in Avio Aero turbomachinery.

The workflow is composed of steps. A step has inputs and outputs. Computational workflow editor can benefit from separating these steps into HPC jobs with inputs and outputs, since they can be reused in other workflows later. For example, EnzymeMiner computational step in Fireprot-ASR workflow[2].

1.2.3 End-Users

The beginning of the thesis introduction mentions that the editor aims to give scientists the ability to compose their custom computational workflows from computational steps. However, setting up the required parameters, such as memory or cores requirements and other parameters described in the TOSCA section3.2, may require knowledge, which only some possess. Therefore, the demand for people with advanced knowledge of the system

[1] Loschmidt Laboratories conduct interdisciplinary research in the field of protein engineering.[1]
[2] The EnzymeMiner is also another tool backed by the Loschmidt laboratories itself.

is increased. Therefore, the scientists were considered as end-users of the editor. They may have a fundamental knowledge of computer science. The role of a workflow architect involves creating tasks with appropriate parameters for software execution. This requires a significant level of computer science knowledge as well as expertise in the target computational systems. The workflow can be later reused by the users with less expertise to compose similar computational workflows.

Chapter 2

Scientific Workflow Tools

Tools and editors for creating workflows already exist, but not all are suitable for the mentioned set of issues. For example, The LifeWatch - Tesseract (section 2.4) belongs to the tools created more often for software deployment rather than for managing scientific workflows. However, not all of the tools described in this chapter can work with scientific workflows. Even though OpenStack - Template Generator (section 2.3) is based on the same problematic as the targeted scientific workflow editor, it cannot work with scientific workflows.

2.1 Alien4cloud

Alien4Cloud is a designer for the Yorc [7] orchestrator by Atos. They are i the Ystia Suite[27]. Multi-Tier infrastructure can be designed in Alien4Cloud, deployed and managed on any cloud and even the HPC schedulers. The graphical interface itself is more suitable for knowledgeable users. However, all requirements for LEXIS platform are covered there.

The Alien4Cloud is a significant source of inspiration for the targeted scientific workflow editor. It is already a part of the LEXIS system and workflow designers use it. The LEXIS project has already solved data transfers, authentication, authorisation and deployment issues. Thanks to the modularity of the Yorc orchestrator, the plugins can extend the functionality and bring new special types of nodes. For instance, datasets transfers into the cloud virtual machines from DDI[1] (section 3.3.1).

The Yorc DDI plugin[26] provides TOSCA components with jobs to handle transfer between the cloud job and DDI, cloud job and HPC job, and HPC and DDI. The data are mounted to the HPC with SSHFS[2] protocol. The HPC tasks computes with data stored in the scratch directory. [3]. LEXIS uses HEAppE (see section 3.3.4), hence the HEAppE Yorc plugin[5] was developed. In the case of a cloud task, Docker containers access fetched datasets by mapping the directory with a dataset from the host virtual machine to the Docker container.

[1] DDI – Distributed Data Infrastructure

[2] SSHFS, as the documentation in code repository mention [21], is network filesystem client based on SFTP [35] protocol and FUSE library. FUSE (Filesystem in Userspace [17]) is an interface for userspace programs to export a filesystem to the Linux kernel.

[3] The scratch directory is supposed to contain temporary files for compute jobs

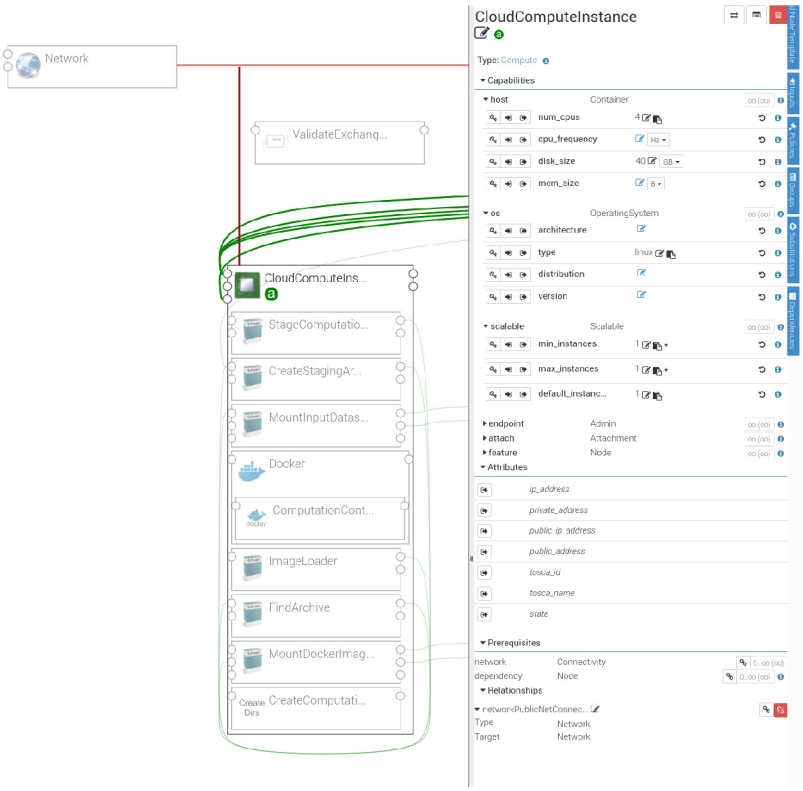


Figure 2.1: Ystia Suite – Alien4Cloud
Source: Own screenshot of application

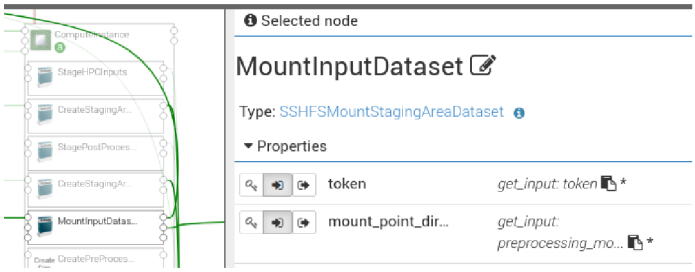


Figure 2.3: Alien4Cloud – Dataset Mounting
Source: Own screenshot of application

The OICD plugin[18] takes care of authorisation and authentication using the OAuth 2.0[4] protocol. Since security is an essential topic in terms of HPC computing, authentication and authorisation should be ensured in all operations, such as scheduling the job on the HPC scheduler or fetching data from DDI.

[4]OpenID protocol [19]

2.2 occiware – TOSCA-Studio

It is a part of DevOps[5] for deploying Multi-Tier application infrastructure. It is well suited for service-oriented architectures. Particular nodes, software, or networking is described in TOSCA format (section 3.2). The graphical user interface part visualises dependencies between the nodes (e.g. task, decision, virtual machine, network). The graphical editor itself runs on the Eclipse development platform. The dependency graph is detailed in the figure 2.4. The TOSCA-Studio [31] is an application based on the Java OCCIware framework.

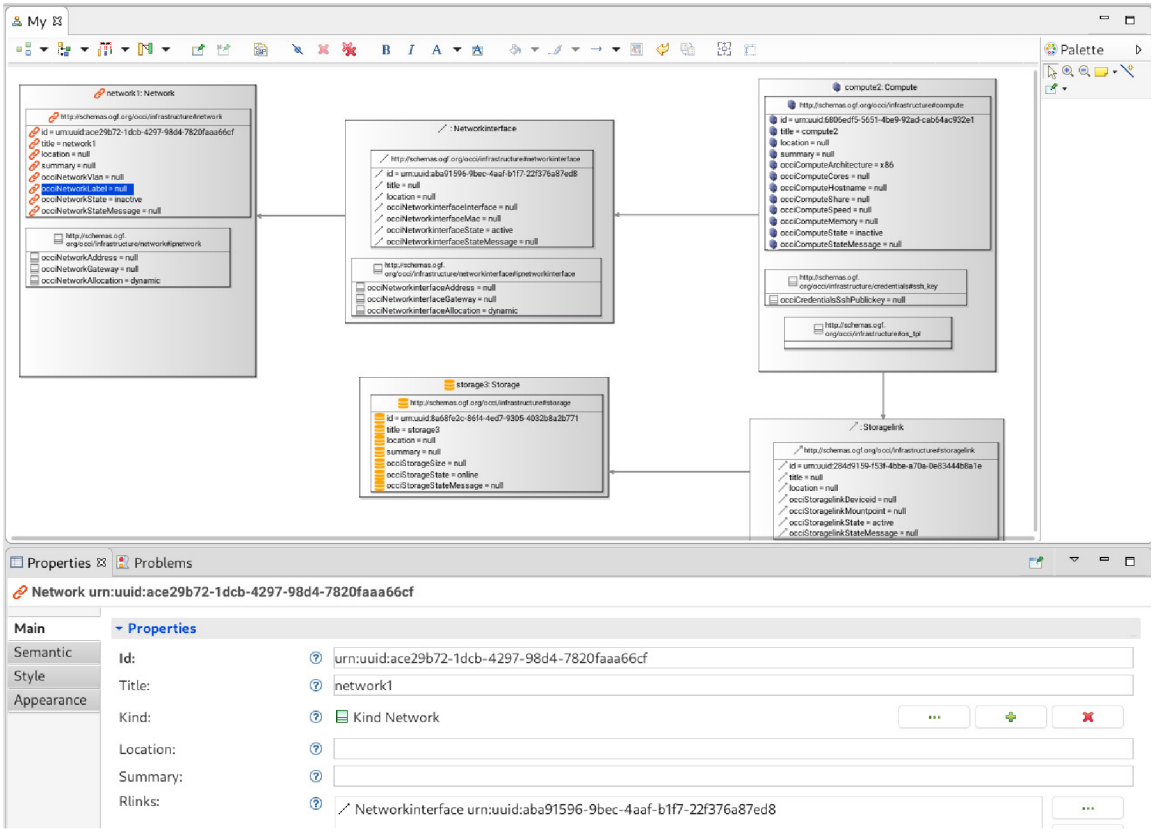
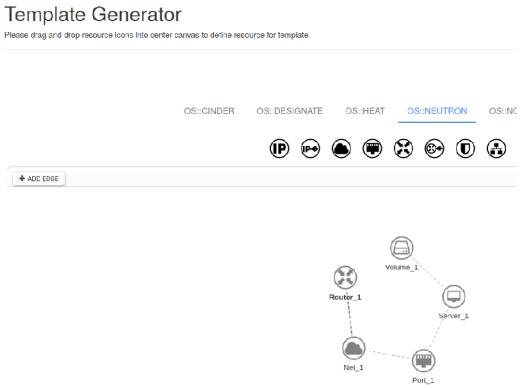


Figure 2.4: occiware – TOSCA Studio
Source: Own screenshot of application

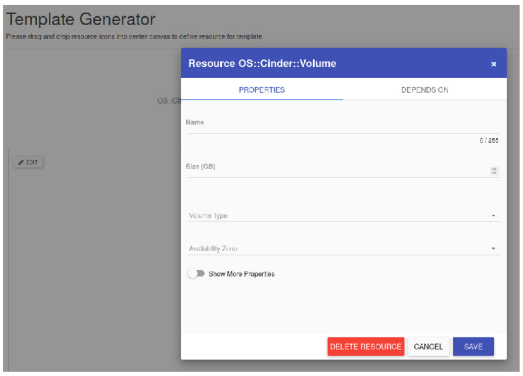
2.3 OpenStack – Template Generator

Alongside the other features provided by OpenStack (section 3.3.5), it offers a stack orchestration feature. It enables users to specify the OpenStack sources in a format called HEAT. HEAT templates are similar to the TOSCA templates. The HEAT template defines topology, relations and specification of sources similar to the TOSCA template. The graphical interface has a minimal visual representation of the dependency graph. The modal window modifies the properties of the particular nodes.

[5]DevOps combines development (Dev) and operations (Ops) to unite people, process, and technology in application planning, development, delivery, and operations. [23]



(a) Relation Diagram
Source: Own screenshot of application



(b) Modal Window

Figure 2.5: OpenStack - Template Generator
Source: Own screenshot of application

2.4 LifeWatch – Tesseract

The LifeWatch - Tesseract is solely for the scientific workflow graphical editing. The components of the dependency graphs are data collection, data processing tasks, and data analysing tasks. Relations between components can be established by clicking on the ports of the source and destination components. However, it is not particularly detailed, but interaction with the graph has basic operations like creation and removal. Even more, the interconnection of the outputs and inputs of the component is provided.

Figure 2.6: LifeWatch – Tesseract
Source: Own screenshot of application

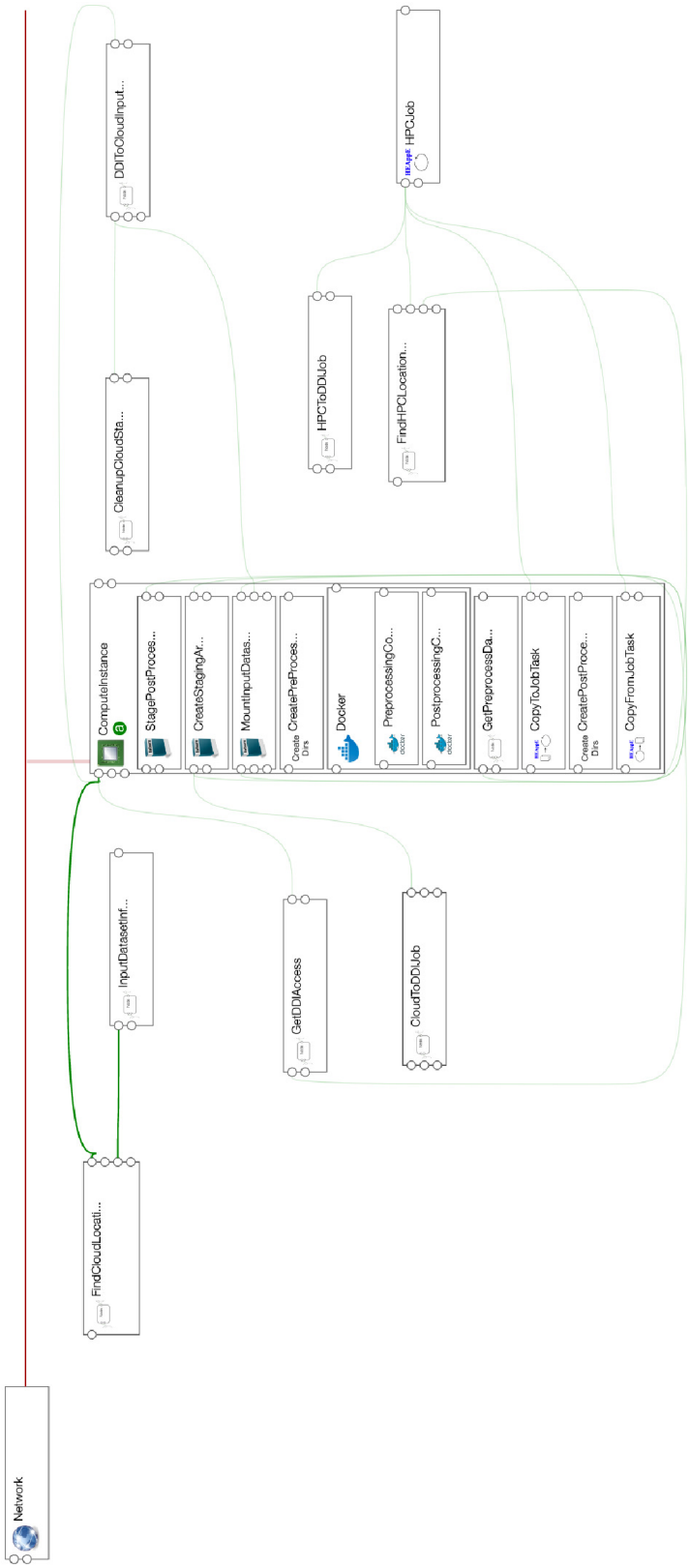


Figure 2.2: Alien4Cloud – topology
Source: Own screenshot of application

Chapter 3

Analysis

3.1 Algorithms and Data Structures

The elementary unit of the targeted scientific workflow editor is a computational task. Interaction with the logic inside the task is ensured with input variables and output variables. The variables can store plain data types like number or string. The task may require some datasets at the input and can also produce a dataset as a result. Furthermore, dependency at plain-data output of other tasks should be taken into account. The most suitable abstraction of all of the aforementioned issues is a non-cyclic directional graph. The abstraction is well described in the article „Characterisation of Scientific Workflows“ [30]. To ensure not having any loops within a graph, the algorithm of loop detections in a graph can be used. The cycles can be found by the depth search algorithm described in [38, Chapter 7 - Directional Graphs]. Considering some edge cases, standalone tasks may occur without input and output connections. They are not a part of the graph, although there is no reason not to include them in the final topology.

3.2 TOSCA - Topology and Orchestration Specification for Cloud Applications

OASIS (Organization for the Advancement of Structured Information Standards) specified a TOSCA standard to describe the topology of cloud-based web services, relationships and their management processes. TOSCA language is based on the serialisation standard YAML[1].

A subset of sections in topology established by TOSCA:

- **inputs** – particular inputs can have a type, description and validation constraints
- **node_templates** – properties with node types like „tosca.nodes.Compute“, cpu count, disk or memory and etc.
- **worflows** – description of imperative or declarative workflows typically for deploying, starting and undeploying topology
- **outputs** – defines output data e.g. server IP address

[1] YAML - Human readable data-serialisation language [25]

Nodes may need to share their properties and attributes like mentioned IP address with syntax construction `{ get_attribute: [ db_server, private_address ] }`. To connect the input parameters to the appropriate node property, the following syntax can be used `{ get_input: db_server_num_cpus }`.

3.3 LEXIS Platform

As mentioned in the introduction, scientific workflow editor will be used within the LEXIS platform. Therefore, this chapter describes some relevant parts of the LEXIS project[14]. The LEXIS project resulted in the construction of a distributed HPC infrastructure to converge big data and HPC. The aim was to build an advanced architecture for big data analysis and High-performance computing applications utilizing modern technologies from HPC to Cloud computing. LEXIS provides ready-to-be-used HPC infrastructure that offers HPC-as-a-Service capabilities without incurring performance/efficiency penalties. More detailed description of the LEXIS platform can be found in the public documentation [16].

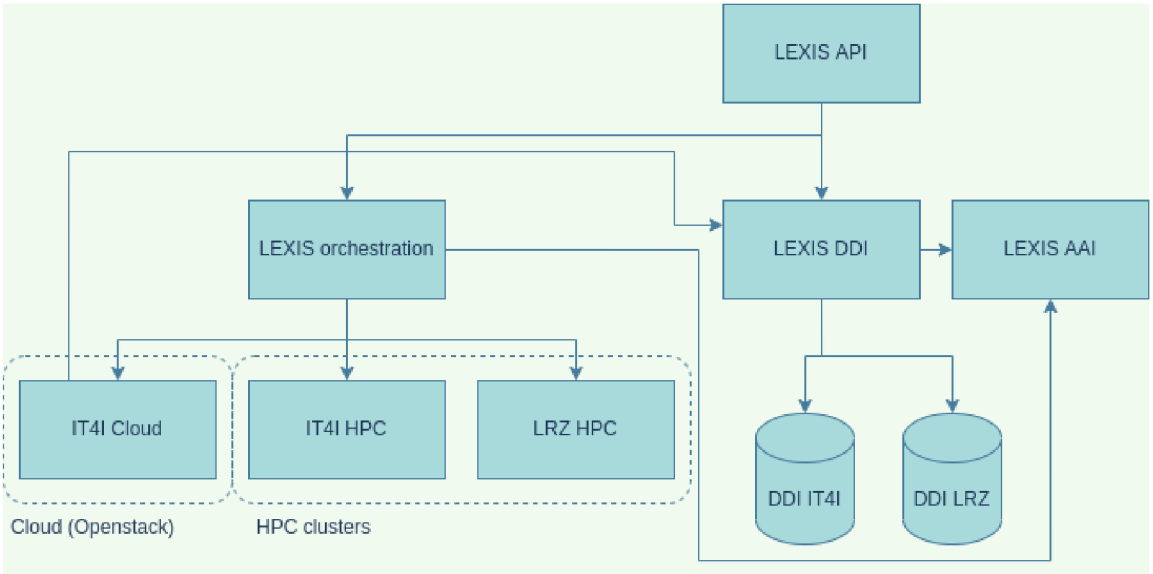


Figure 3.1: LEXIS Infrastructure
Source: Own diagram

3.3.1 LEXIS DDI

Data storage federation and data management were the main goals the LEXIS project faced. The elementary unit stored on the DDI is a dataset. Datasets could be inputs or outputs of computational workflows in the LEXIS. They created a system based on iRODS (The Integrated Rule-Oriented Data System)[8] and the system integrates EUDAT's European research data services.

The iRODS solution guarantees that a unified logical file space is created and accessible to all participants. The rule and event orientation of iRODS allows us to write routines that react to events such as data ingest, enforcing policies relating to data distribution, rights management, and reduplication.

3.3.2 LEXIS orchestration

During the LEXIS project, an orchestration system was developed with the capability to execute complex workflows that involve a combination of HPC, Cloud computing and Big Data tasks. To serve as the orchestrator, Yorc [7] was chosen - an open-source TOSCA orchestrator that seamlessly integrates hybrid cloud and HPC capabilities. Yorc is also workflow-driven and allows fully customisable applications behaviour. For interaction with Yorc, there is a comprehensive REST API. The Yorc was originally developed by Bull Atos Technologies. The alien4cloud TOSCA application designer manages the creation of applications. The alien4cloud also manages the deployment of applications via Yorc to the Cloud computational resources and HPC clusters. The workflow architects can arrange topology consisting of nodes with software and types, which are deployed to the selected infrastructure. The workflow architects have extensive knowledge of the system and the infrastructure.

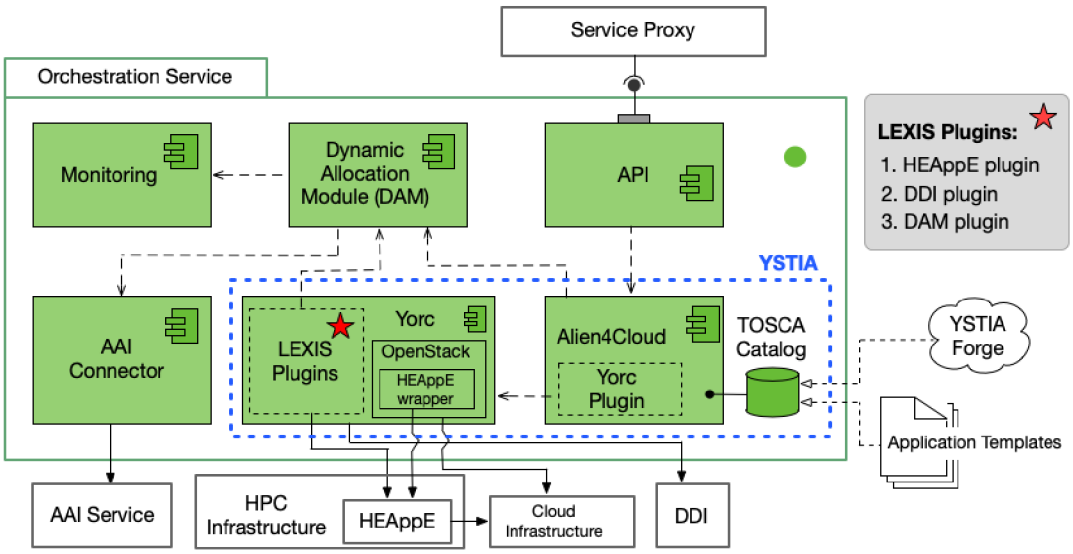


Figure 3.2: LEXIS Orchestration
Source: LEXIS D4.6 p.6 [36]

3.3.3 Apache Airflow

As stated on the web presentation [2], the Airflow is a platform created by the community to programmatically author, schedule and monitor workflows. The Airflow platform is scalable, dynamic and easily extensible. The workflows within the Airflow are defined in Python language with the use of built-in operators or custom ones. Nowdays, the LEXIS platform uses Apache Airflow aside the Yorc orchestrator and Alien4Cloud. The Airflow is extended with custom plugin for authorization and with a set of custom operators for communication with the rest of the LEXIS infrastracture services.

3.3.4 HEAppE - High-End Application Execution Middleware

HPC-as-a-Service is a middleware that facilitates the execution and management of jobs on HPC clusters, as well as the collection of information related to those jobs. This middleware provides a REST API that enables users to submit computations or simulations on HPC

infrastructure via HEAppE, which can also monitor the progress of the job and notify the user if needed. Applications should be installed before using them on the cluster by an authorised person, in the case of IT4Innovation's clusters.

The jobs may require input datasets. The datasets can be provided to the jobs in scratch directory. To transfer the files into the directory, the HEAppE provides a temporary SSH key.

3.3.5 OpenStack

OpenStack is a comprehensive platform designed for managing a collection of interconnected components, such as compute pools, networking, and storage resources. It provides a web GUI and terminal utility to manage the sources and services. User can set up limits and quotas for sources like IP addresses, CPUs, or number of compute instances. The platform also offers various operations with storage, like taking snapshots.[44]

Chapter 4

Application Design

The application for designing computational workflows should provide a graphical interface that accommodates two levels of detail for end-users with varying levels of expertise, including non-computer scientists and workflow architects. The application should communicate via a secure private network connection, taking into account the sensitive nature of the information involved and the need for trustworthy communication with LEXIS services. Therefore, only the graphical interface is exposed to the users. A part of the editor is the registry with REST API for storing metadata about the cloud images (Docker images). The graphical interface is aggregated in the .NET Blazor application together with the TOSCA editor interface. The tight integration between the .NET MVC structure and the rendered pages can help address the challenge of generating TOSCA templates.

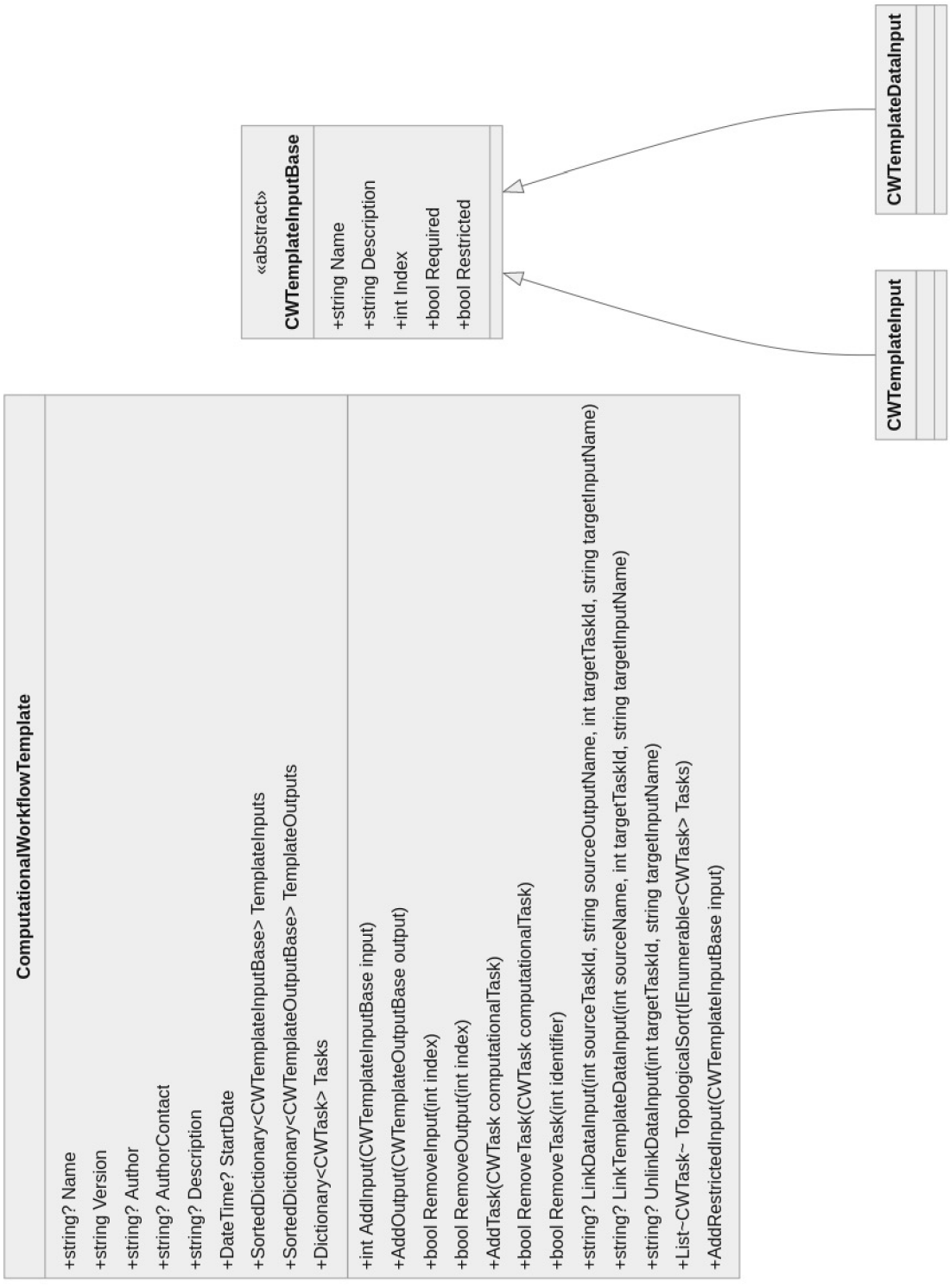


Figure 4.1: Application Design – Internal Representation of Topology – Class Diagram (part 1)

Source: Own diagram

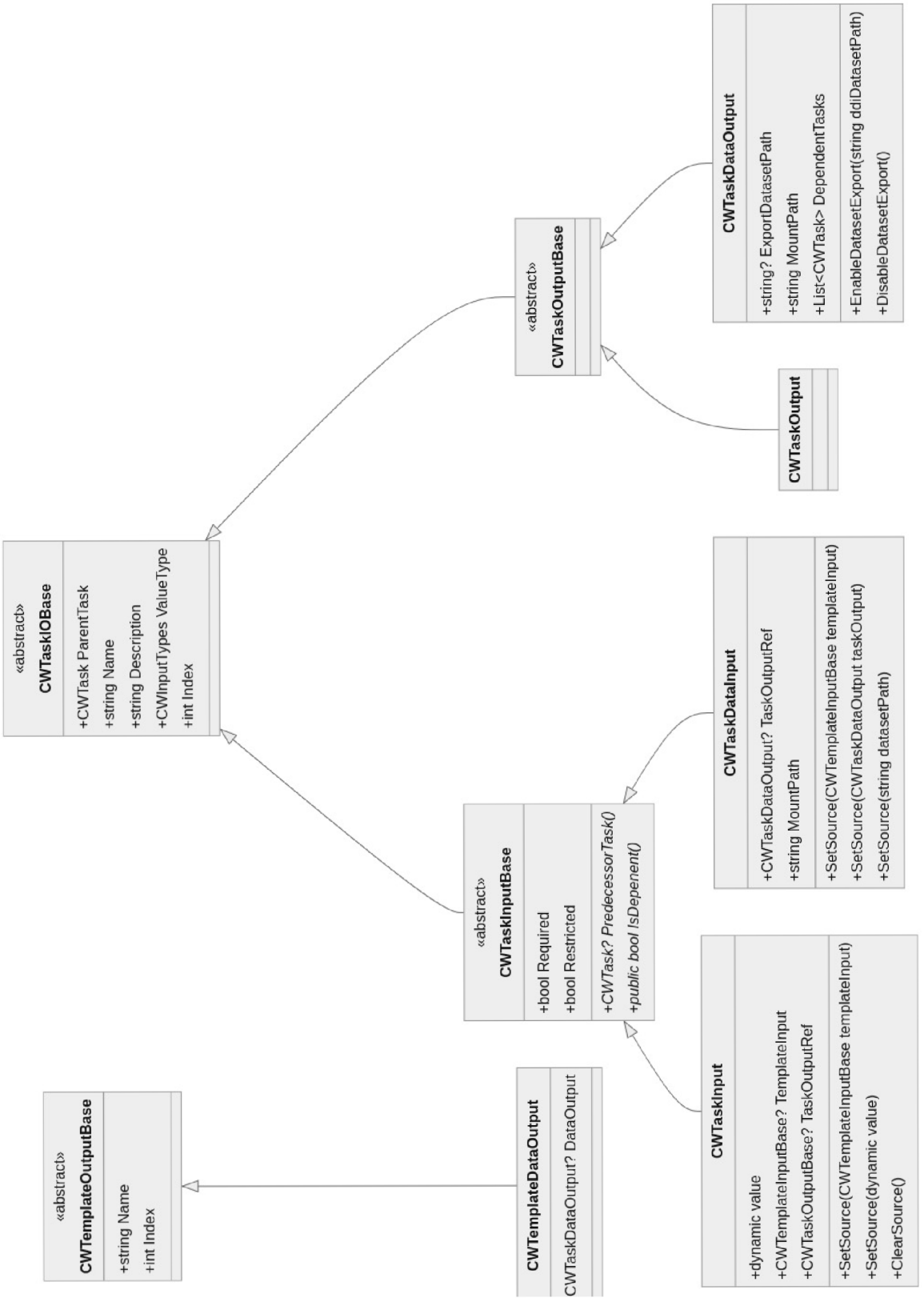


Figure 4.2: Application Design – Internal Representation of Topology – Class Diagram (part 2)

Source: Own diagram

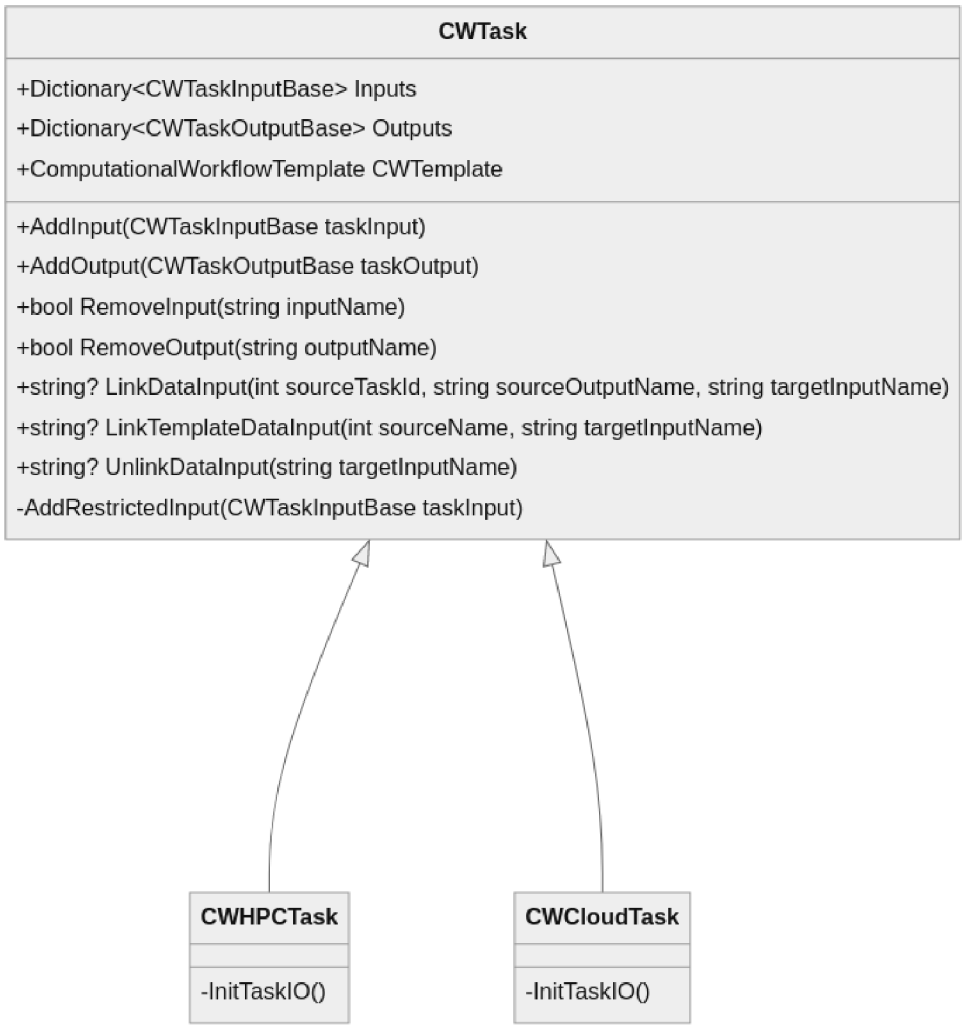


Figure 4.3: Application Design – Internal Representation of Topology – Class Diagram (part 3)
Source: Own diagram

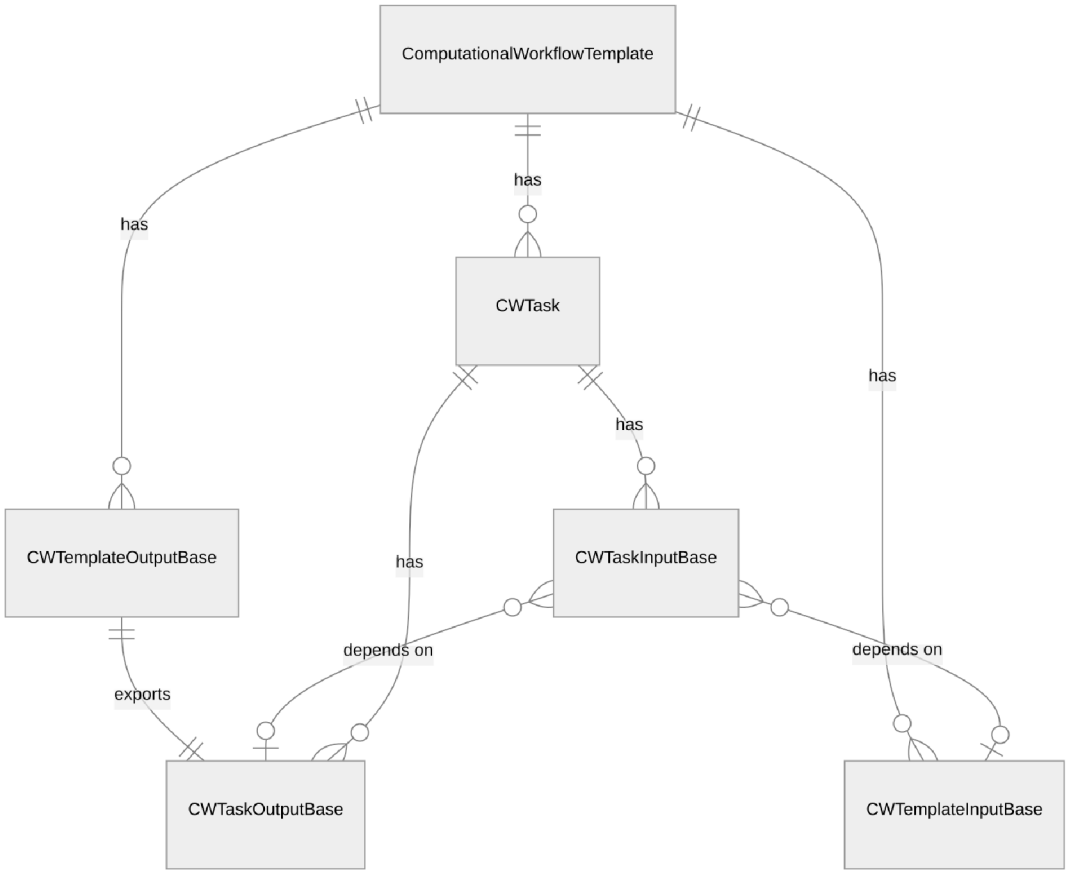


Figure 4.4: Application Design – Internal Representation of Topology – Relation Diagram
Source: Own diagram

4.1 Internal Representation of Workflow Topology

To provide a comprehensive internal representation of the topology, we propose to utilize classes and their relations. The classification is based on the workflow structures illustrated in figure 5.1. in the article „Characterisation of Scientific Workflows“ and other advanced workflow topologies. The article describes basic workflow types: process, pipeline, data distribution, data aggregation and redistribution. The workflow presented in section 1.2.1 for Avio Aero turbomachinery is a typical example of a basic computational workflow. The classes in the following list are visualised in the figure 5.1.

- Input (of the computational workflow)
- Output (of the computational workflow)
- Task
  - Cloud task

  - HPC task
- Task requirement
- Task input
  - Task data input
- Task output
  - Task data output

Computational Workflow Input

The input for a computational workflow can be in the form of a string data type that is acceptable by the corresponding task. It is possible for a certain input to carry a dataset UUID [39] that is entered by the user. Such inputs can be attached to the task data input for further use in the workflow.

Computational Workflow Output

The output of the computational workflow has the same types as the input, but datasets, a particular kind of output, are treated differently. In this case, the creation of the dataset in the DDI is expected after the computational workflow execution finishes.

Task

A task defines computational tasks executed on an HPC cluster or a Cloud. It depends on the user's choice. An HPC cluster task could be chosen from the list of available tasks suggested by the HEAppE API. The HEAppE also defines task inputs. Cloud tasks are similar, but the cloud task could be chosen from the list of registered cloud images in the cloud images metadata registry. In case a task's data input is related to another task's output, implicit data transfer should be generated.

Task requirement

Task requirements are internal abstractions that represent dependencies between task inputs and outputs and other tasks. However, these dependencies cannot form a loop.

Task Input and Task Data Input

To ensure flexibility in task inputs, the editor should allow the user to specify a wide range of input types, such as strings, integers, and UUIDs for dataset inputs. In addition, each task should be able to receive data inputs from other tasks within the workflow. The user should be able to specify the source of the data and the input name, which will then be used to retrieve the data in the task implementation.

Task Output and Task Data Output

For task data outputs, the editor should also allow the user to specify a range of output types. These can include strings, integers, and UUIDs for dataset outputs. Similar to task inputs, task outputs can also be the data input of other tasks in the workflow. The user

should be able to specify the output name. The data from the task output data can be for downstream tasks later.

4.2 Cloud Image Metadata Registry

The purpose of the registry is to hold information about Docker images, which do not contain necessary information such a input environment variables and input and output paths to the directories inside the instance container, where mapped data from the datasets should be or where outputs of the tasks can be found. Thus, it is necessary to store the metadata. Otherwise, correct functionality of the Docker containers cannot be guaranteed. The metadata is planned to be stored in a database with operations accessible via REST API.

4.3 GUI/UX Graphical Design

The graphical interactive prototype was inspired by Alien4Cloud and other editors mentioned in section 2. The application should provide GUI for managing cloud images in the registry (section 4.2) and creating as well as editing workflow templates. In addition to providing dashboards for workflow templates and cloud images, the GUI's editor with a flow diagram is a critical component. The flow diagram should provide a representation of the workflow, including nodes that represent both cloud and HPC resources as well as nodes for the workflow's data inputs. The diagram should be designed to allow users to easily visualize and understand the flow of the workflow, and should provide intuitive mechanisms for linking nodes to indicate data input and output dependencies of the computational tasks. By interviewing possible users, the following changes were proposed and accepted:

- Separate the navigation menu (on the left) from the action menu (on the right).
- Outline the currently active page in the navigation menu.
- Choose more suitable icons for actions.
- Create buttons with help for the forms.
- Use the Docker naming terminology in the cloud image forms.

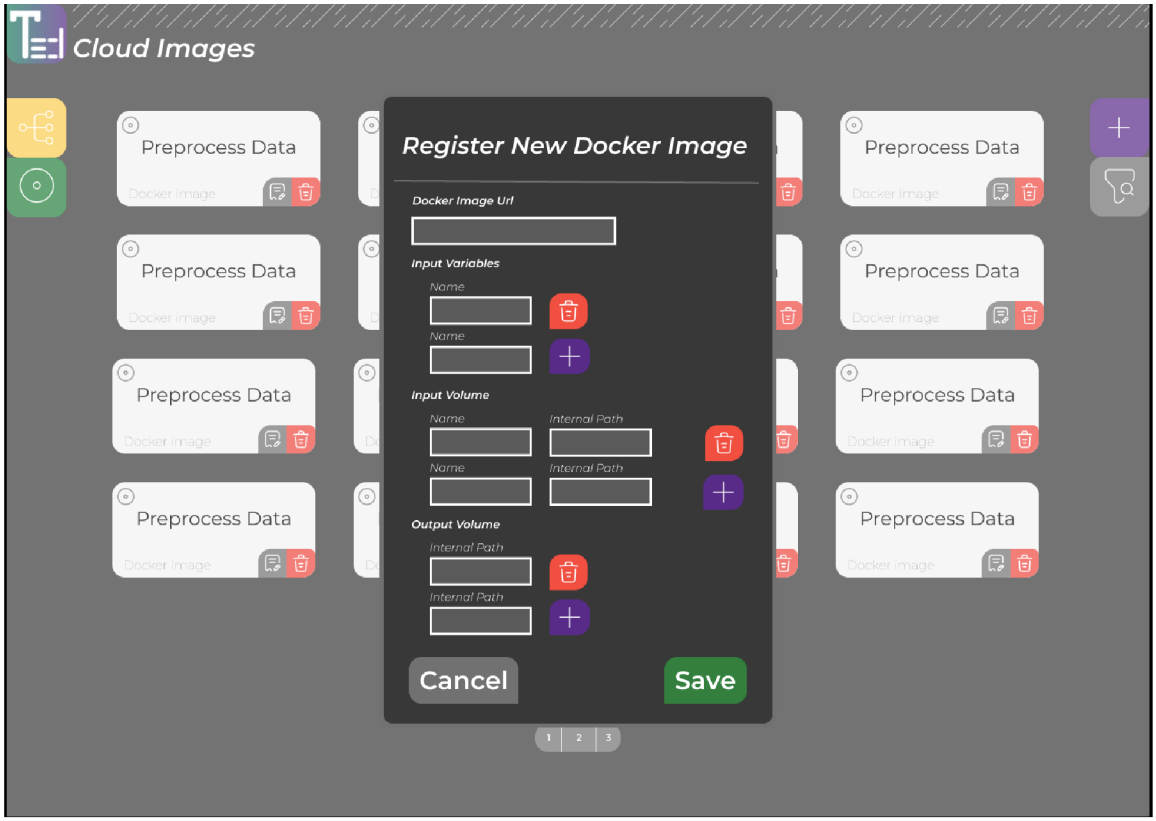


Figure 4.5: GUI Design – Register New Docker Image
Source: Screenshot from Figma design tool

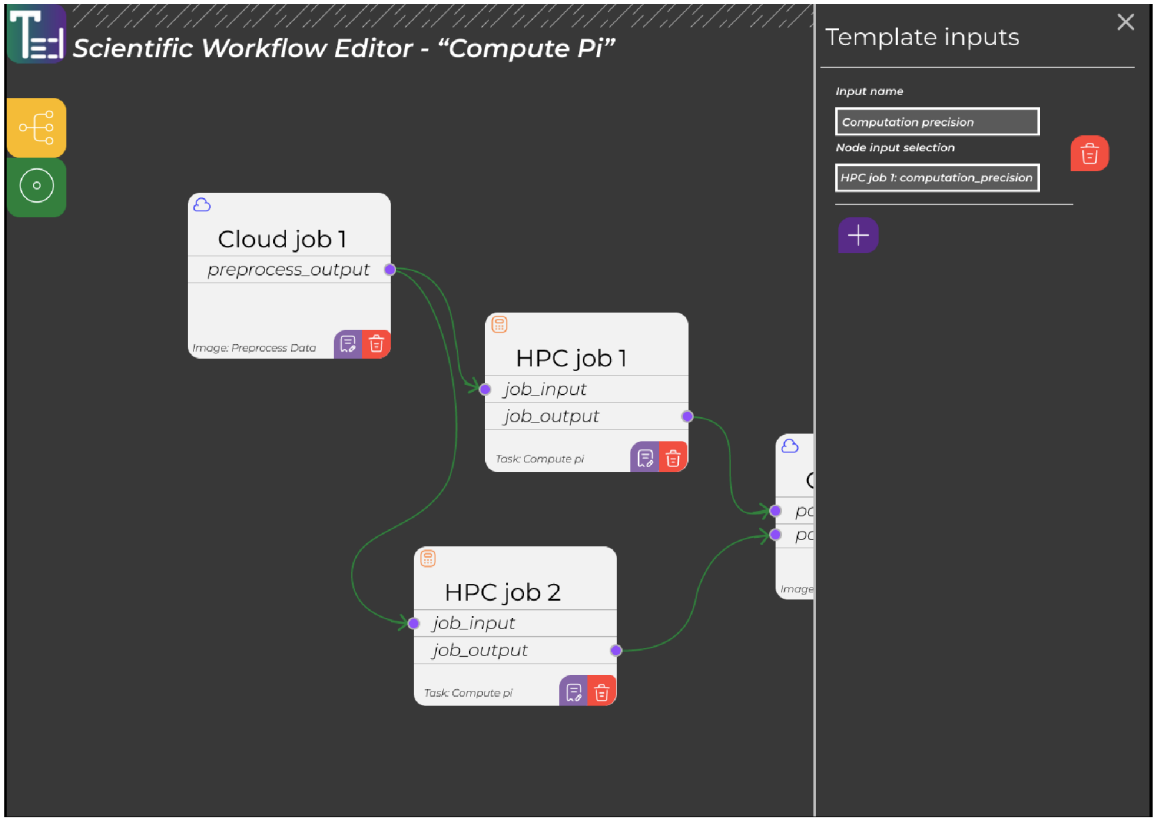


Figure 4.6: GUI Design – Workflow Inputs
Source: Screenshot from Figma design tool

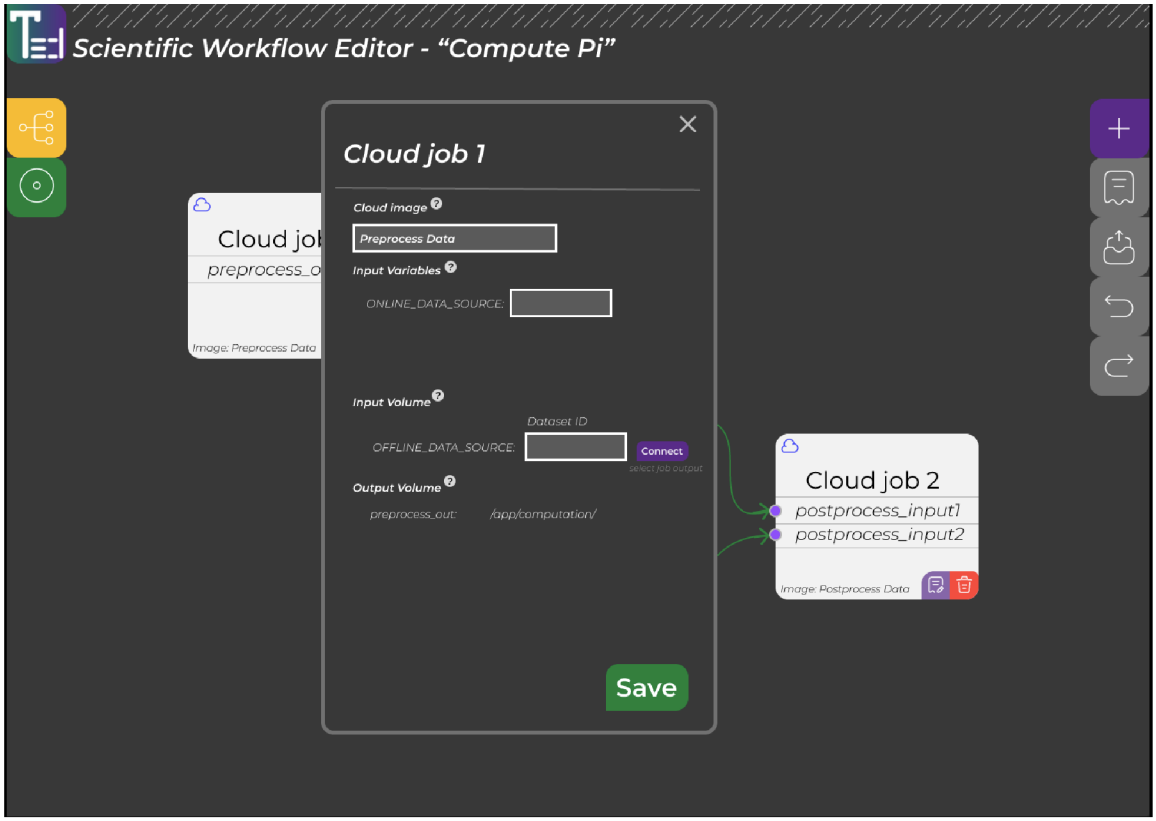


Figure 4.7: GUI Design – Modify Cloud Task Details
Source: Screenshot from Figma design tool

4.4 Advanced features

During the discussion, several advanced features were identified for the workflow editor. One interesting feature is the ability to statically specify particular execution locations. This feature can be particularly useful for users who require a specific HPC cluster for orchestrating their application. However, in some cases, users may need to specify the execution location before the workflow execution, or even the number of instances for a computation. To address these situations, a templating engine can be integrated into the workflow editor, allowing users to define their own templates and specify the required parameters before execution.

Chapter 5

Implementation

The particular implementation of the graphical editor for composing computational workflow is explained in the following section. According to the design in section 4.1, detailed implementation of entities like *computational workflow* or task's data inputs is described further. The implementation takes advantage of object-oriented programming and imparts the software generality and extensibility. The TOSCA workflow description is not the only one on the market. Thus the editor may consider using different formats like Heat Orchestration Template [1] used by OpenStack. Given the graphical editor's emphasis on user-friendliness, it is essential that the editor's state is both persistent and portable. However, any of the target specifications do not guarantee to have enough information for backward conversion. Therefore, a custom format without any information biased by target technology is introduced in section 5.2.6.

5.1 Used Technologies

The implementation of the graphical user interface is accomplished using the .NET framework in C#, in conjunction with the JavaScript library JointJS for diagramming (see the website [10]). I chose the .NET Blazor Server framework [40]. It directly competes against the contemporary JavaScript frameworks like React [29]. The framework uses Model-View-Controller architectural pattern. Razor syntax allows for the description of HTML components and the associated code that pertains to each component. However, when compared to Razor Pages[2], the Blazor Server framework allows for interaction with the application without the need to reload the web page, as is common in modern web applications. One of the practical features offered by the framework, which greatly benefits the editor implementation, is the interaction between the browser and Blazor Server via a WebSocket. As described in section 5.4.1 on task diagram implementation, the editor can invoke JavaScript and C# functions bidirectionally. The function invocation requires data serialisation described in section 5.2.6. REST API implementation is not necessary. Thus, the editor implementation does not require the overhead of communicating with REST API.

Emitting the internal representation to the final TOSCA format requires a basic .Net package *YamlDotNet*. However, TOSCA syntax contains some repetitive, more nested structures. The ToscaDOM library by Ing. Jiří Dvorský, PhD, proposes a practical level of

[1] Detailed specification of the Heat Orchestration Template is described on website [6]

[2] .NET Razor Pages use a simplified web application programming model

abstraction into classes, which helps to implement the functionality much more efficiently. The library is written in C#.

Referring to the dependency diagram mentioned in section 4.3, it constitutes the main visualization component of the application. While there are several libraries available on the internet, such as the Blazor.Diagram [3] library, the JointJS library [10] was chosen to ensure the editor's sustainability. This well-documented JavaScript library has both a community and professional version available.

5.2 Computational Workflow Core

To ensure the clarity of the diagram in figure 5.1, more details about the entities are provided in this section. The abstraction is described below using the top-bottom approach. The code with workflow core in C# is in file *ComputationalWorkflow.cs* covered in namespace *ComputationalWorkflow*. On the other hand, the JavaScript class abstraction is located together with the graphical part implementation in file *WorkflowEditor.razor.cs* (the graphical part implementation is in section 5.4.1).

5.2.1 Computational Workflow Template

Blazor Server

The term *computational workflow template* comes from the Tosca specification, but in the abstraction, it makes more sense to describe it as a computational workflow. The top-level entity that contains all other workflow entities is named *ComputationalWorkflowTemplate*. The entities are stored in *SortedDictionary* data type to keep them in order and quickly accessible. It controls addition and deletion of computational workflow inputs, outputs and computational tasks. It is necessary to assign unique identifiers to all entities, so the responsibility for this operation is assigned to the class. Otherwise, it will not be possible to refer to them. Due to serialisation restrictions, the identifiers are essential for operation cross-invocated from both JavaScript and Blazor Server framework (see section 5.2.6).

The class *ComputationalWorkflowTemplate* exposes methods for linking sources of data inputs. Valid sources for data input are computational workflow data input (see section 5.2.2) or task data output (see section 5.2.5). Arguments of the method are integers and strings to simplify invocation from JavaScript.

JavaScript

A similar lightweight abstraction of computational workflow is demanded to keep a consistent state between the diagram rendered by JavaScript and the state controlled by *ComputationalWorkflowTemplate* class. The abstraction is covered in class *WorkflowEditor*, which also manages the JointJS [10] diagram instance (see section 5.4.1). The class *WorkflowEditor* exposes methods for:

- Adding and removing computational tasks
- Adding and removing computational workflow's data inputs
- Linking and unlinking data inputs and outputs of tasks or computational workflow's data inputs
- Enabling and disabling export of computational task's data outputs
- Name modification of computational task
- Name modification of computational workflow's data input

All of the mentioned operations are invocated when a user interacts with menus controlled by Blazor, and some diagram changes are requested.

5.2.2 Computational Workflow Input

Computational workflow's input may be, for example, number, string, date or dataset from LEXIS DDI [3].

Blazor Server

Both types of inputs have similar properties. They are name, description, value type (number, string, date, etc.), flag required, flag restricted and index for identification. Therefore, an abstract base class *CWTemplateInputBase* implements it. However, considering the base class's generality, the base class does not have a property for holding a default value. Instead, the class *CWTemplateInput*, which inherits the base class, has the *DefaultValue* property. The default value is not mandatory. The value of template input is not considered because the user specifies it just before the computational workflow execution instead. The default value has *dynamic* data type. The *dynamic* is a static type in C#, but an object of type *dynamic* bypasses static type checking. The *dynamic* data type supports any operation. Nevertheless, the consequence is that the compiler does not check, the method call, but the error occurs in runtime to give you an example. Instead of *dynamic* data type, a class templating could be used. However, it demands a supplementary base class implemented because it is often required to distinguish input from data input. Currently, the class *CWTemplateInputBase* has only *CWTemplateDataInput* and *CWTemplateInput* derived classes. Thus, the class of the instance of input or data input is identified by checking `input is CWTemplateDataInput` or `dataInput is CWTemplateInput`. By using class templating, more comparisons will need to be considered.

The default value of template data input can be validated against LEXIS DDI, however, this validation is not included in the current implementation. Nested input structures need to be considered in further development because the implementation demands a more complex graphical interface to be implemented, and ToscaDOM still needs to support it. A form in the input's menu restricts user inputs, thus, is not implemented in any of the classes of the computational workflow abstraction. Menu and validation is described in greater detail in the 5.4.2 section.

JavaScript

Keeping the class abstraction lightweight, the class *TemplateDataInput* contains only methods to get an identifier to manage the node in JointJS [10] diagram, set a name of the input and some other graphical interface related methods (see section 5.4.1). The identifier is generated from the index given to the *CWTemplateDataInput* when adding data input to the computational workflow via *AddInput* method of the *ComputationalWorkflowTemplate* class.

[3] DDI - Distributed Data Infrastructure with REST APIs for storing data and controlling their transfers between distributed infrastructure

5.2.3 Computational Task

A computational task represents computation on an HPC cluster or cloud. A computation may have input parameters such as precision. Typically, a computation may compute on a data set, creating an output data set. The output data set may be used later in different task computation in the computational workflow. A user can set a default value for the data input or connect the input parameter to the overall computational workflow inputs.

Blazor Server

Implementation of the class *CWTask* not only contains the dictionaries with the task inputs and outputs but also methods for addition and removal of them. The code outside the class does not access the dictionaries directly when adding or removing the inputs and outputs. Therefore, there are methods for it:

- *AddInput* Adds class *CWTaskInputBase*, the base class of task inputs, sets the restriction flag to false. Adds unique integer identifier to the added task input. (see section 5.2.4)
- *AddOutput* Adds class *CWTaskOutputBase*, the base class of task outputs.
- *AddRestrictedInput* Adds class *CWTaskInputBase*, the base class of task input, sets the restriction flag to true. The method is protected from external use.
- *RemoveInput* Removes task inputs from the dictionary by their name. It is only possible to remove non-restricted inputs externally.
- *RemoveOutput* Removes task output from the dictionary by their name.

There is an additional class property *Location*. It represents a preferred computation location, which a user can set in the editor. The integer *Identifier* property is present for more effective management of computational tasks.

To simplify the synchronisation of the state between JavaScript and .NET, the methods *LinkDataInput*, *LinkTemplateDataInput* and *UnlinkDataInput* are exposed. Especially, the arguments contain only identifiers with atomic types. The methods return a string message in case of an error to give feedback to the caller.

JavaScript

Implemented class *Node* holds task inputs and output similar to the implementation in .NET. Additionally, it implements a method for changing the title, which is displayed in the diagram for better user orientation. It also implements methods with atomic-typed identifiers for cross-invocation simplification. The class also implements the identifier property similarly to the class implementation the C#.

HPC and Cloud Computation Task

The initialisation is the only thing the HPC and cloud computational task differs from the basic class. Its initialisation includes the creation of new instances of restricted (cannot be removed by the user) inputs and outputs. The inputs having the default value or not required inputs (has required flag set to false) do not need to be filled before the computational workflow execution by the user. HPC task-specific inputs are:

- Name – The name of the computational task, which the user can specify and will be displayed in the GUI diagram.
- heappe_uri – URL of HEAppE instance deployed for target computation location. In further development, the URL input could be hidden, and the URL could be filled automatically according to chosen computational location. LEXIS API needs to expose relevant information for the functionality to be implemented.
- Project – LEXIS project short name identifier. It is mandatory for proper accounting of computation in the LEXIS system. The LEXIS or user should provide it before the execution of the computational workflow.
- ClusterId – Cluster identifier specific for the HEAppE implementation.
- CommandTemplateId – Command template is a script installed inside the HPC and registered in the HEAppE instance for the particular computation project. It should be hidden from the user in further development of the editor.
- MinCores – Minimum number of cores requested for the computation. The default value is 1.
- MaxCores – Maximum number of cores requested for the computation. The default value is 128.
- WalltimeLimit – HPC scheduler will stop the computation after the computation reaches the specified limit. The default value is 120 minutes.
- Priority – Job priority for HPC scheduler. The default value is 4.
- ClusterNodeTypeId – Cluster node type identifier is specific for HEAppE. It helps to differentiate the computation cluster with different modules like GPU.
- FileTransferMethodId – Specifies the type of protocol to use for data transferring between the computational node and staging area of DDI.

Some of the mentioned shouldn't necessarily be visible to the user, but hiding them requires more extensive integration with LEXIS. Ideally, the identifiers should be mapped with labels to give the user better awareness of the task configuration.

In comparison with a HPC computational task, the cloud computational task adds just three following task inputs:

- Name – The name of the computational task, which the user can specify and will be displayed in the GUI diagram.
- DockerImageURI – URL of docker image for cloud computation task. The URL is planned to be hidden for the user in further development. The user can then choose from the docker images registered to the docker metadata registry (see section 4.2).
- Labels – Optional labels for advanced handling of cloud computational jobs in Kubernetes[4], which LEXIS uses internally for Cloud computations.

More specifications like cloud instance flavour or module requirements can be introduced later.

[4]Kubernetes is an open-source container orchestration system for automating software deployment, scaling, and management. [12]

5.2.4 Computational Task Input

Computation usually computes on some data with specified precision. However, some inputs may be used for the HPC job scheduler too. Therefore, all the mentioned inputs should have the base class to work with. More about the class abstraction is in according sections below.

Blazor Server

The base class for the computational task input is *CWTaskInputBase*, but because the computational task input and output have similarities, the shared part is implemented in class *CWTaskIOBase*. Both inputs and outputs have a name, description, type of default value and internal identifier comparable to the workflow input class (see section 5.2.2). What the workflow input does not have, and computational task input and output have, is a reference to the parent computational task. The base class *CWTaskInputBase* for computational task inputs also implements the possibility of connecting workflow input as the source of the value. The workflow input can be restricted, and the user cannot remove the task input from the computational task. Also, the input must be filled in before the execution. Therefore, there is *Required* flag with a boolean value.

However, the task input does not have to be sourced only from workflow input. The source could be some computational task, of course. Thus, the chaining of computational tasks is achieved. Currently, only data chaining is allowed between the tasks. The basic task input is implemented by *CWTaskInput* class, and *CWTaskDataInput* class implements data input. When the data input is not sourced from workflow input or another computational task, the DDI dataset identifier should be present. As mentioned, the input can hold the value or reference to some task or workflow input. Thus, both the classes implement method *SetSource*, which handles inner properties to be appropriately set. The *SetSource* method is overloaded and can accept different types of inputs: a value, a workflow template reference, or a task output reference. If a value is provided, it is simply set as the new source, and any existing references are unlinked. If a workflow template reference is provided, then the current source, value, and any other references are unset. Similarly, if a task output reference is provided, the current source, value, and any other references are unset as well. The method *ClearSource* clears references to any source and also unsets the value. Compared with *CWTaskInput*, the *MountPath* holds information about the path, where the dataset or data from the other task will be mounted.

JavaScript

The JavaScript part implements just the data inputs and outputs. Thus, the user can manage data transfers in the graphical interface (see section 5.4.1). The class *NodeIO* implements both input and output. The property Type, with possible string value *input* or *output*, distinguishes the class type.

5.2.5 Computational Task Output

Currently, the application only supports dataset outputs, but it can also handle outputs that hold basic values, similar to the *CWTaskInput* class. The user can perform various actions with the dataset output such as exporting it to the LEXIS DDI, copying or mounting it to another computational task, or sharing it between multiple other tasks. Subject to the support of the LEXIS transfer API.

Blazor Server

The base class *CWTaskOutputBase* does not implement any logic now, but it is there to keep the implementation general, similar to the computational task input class structure. The base class inherits from the class *CWTaskIOBase* (see section 5.2.4). As mentioned before, the implementation includes the support of value-based task outputs, but it is not currently implemented. Therefore, the class *CWTaskOutput* exists. The class *CWTaskDataOutput* for data outputs inherits from the base class. The property *MountPath* has the same purpose as in the *CWTaskDataInput* class. It holds a relative path to the directory in a computational task, which will be the output. When exported to LEXIS DDI, the property carries the string with a path in the iRODS. The method *GetStagingAreaPath* generates a static address for data transfer between the tasks. It is relevant to the way the LEXIS staging area works. To grant safe data transfer, each data directory copied from the task should have its unique path. It is not optimized yet in the current version of the workflow editor, and the section 5.3.4 describes more about the data transfer.

JavaScript

The class *NodeIO* implements the computational task output in JavaScript. For more information, see section 5.2.4.

5.2.6 Serialisation

Although serialisation was briefly mentioned in section 5.1, but a detailed explanation has not yet been provided up to this point. The serialisation explanation is irrelevant without an adequately described computational workflow core (see section 5.2). Serialisation is mandatory to synchronise the state between the C# in .NET framework and JavaScript. The computational workflow abstraction classes should have JSON [5] interpreter. The serialisation to JSON gives the advantage of saving the editor's state to the database. The .NET framework has built-in package *System.Text.Json.Serialization* abstract templated class *JsonConverter*. All the computational workflow classes have implemented their serialisation classes in file *ComputationalWorfkflowSerializer.cs*. Nevertheless, the classes currently implement just the *Write* method. The *Read* method is not implemented yet. The implementation has simplicity and unambiguity as a goal. Some converters delegate conversion to another converter class to maximise code sharing. The following sections briefly describes the result JSON structures.

Computational Workflow

- name – Computational workflow name. It represents the value of the *Name* property
- version – Workflow template version. It represents the value of the *Version* property
- author – Computational workflow template author. It represents the value of the *Author* property
- authorContaxt – A contact to the author. It represents the value of the *AuthorContact* property
- description – Computational workflow name. It represents the value of the *Description* property
- startDate – When the date is specified, the workflow will be started at the specified date and time. It represents the value of the *StartDate* property
- templateInputs – List of templates inputs

Workflow Input

- identifier – An unique identifier of the workflow input. It represents the value of the *Identifier* property.
- name – Name of the workflow input. If it is not restricted, then it can be modified by user. It represents the value of the *Name* property.
- description – Optional description of the workflow input. It represents the value of the *Description* property.
- valueType – The data type enumeration held by the property *ValueType*. The converter converts the enumeration to a string value.
- required – The flag for the required input should be filled before the workflow execution. It represents the value of the *Required* property.

[5] JSON - JavaScript Object Notation [45]

- restricted – Flag for the input, which the system requires. The user cannot remove it. It represents the value of the *Restricted* property.
- default – Default value. It represents the value of the *DefaultValue* property.

Workflow Data Input

The workflow data input differs from the workflow input just by the possible type of the default value. In the case of the data input, it is a string type because it holds the path to the dataset in LEXIS DDI (iRODS).

Computational Task

- type – Specify the type of the computational task. The possible values are *hpc*, *cloud* or *unknown*.
- identifier – Unique identifier of a computational task in the computational workflow. It represents the value of the *Identifier* property.
- location – Represents a preferred computation location, which the user can set in the editor. The *Location* property in C#.
- taskInputs – List of computational task inputs.
- taskOutputs – List of computational task outputs.

Computational Task Input

- type – In the case of task value-based input, it holds string value *taskInput*.
- name – Is equivalent to *Name* property.
- description – Is equivalent to *Description* property.
- required – Is equivalent to *Required* property.
- restricted – Is equivalent to *Restricted* property.
- predecessorTask – The parent task identifier of the output, which is the source of the input. It can be undefined.
- ref – The name of the output, if the *predecessorTask* is defined. When the *predecessorTask* is undefined, it is the name of the computational workflow input. Otherwise can be undefined, meaning that the input is value-based (the input is the source of the value). Compose the properties *TaskOutputRef* and *TemplateInput* implement in core abstraction.
- value – The value of the input. It is defined when the input has no other source. It is equivalent to *Value* property.

Computational Task Data Input

It is equivalent to the computational task input. The list below mentions differences only.

- type – In the case of task data input, it holds string value *taskDataInput*.
- mountPath – Is equivalent to the *MountPath* property. The path where the input data should be mounted.

Computational Task Output

- type – In the case of task data input, it holds string value *taskOutput*.
- name – Is equivalent to *Name* property.
- restricted – Is equivalent to *Restricted* property.

Computational Task Data Output

- type – In the case of task data input, it holds string value *taskDataOutput*.
- name – Is equivalent to *Name* property.
- restricted – Is equivalent to *Restricted* property.
- exportDatasetPath – Is equivalent to *ExportDatasetPath* property. It is undefined when the data are not exported to the DDI.
- mountPath – Is equivalent to the *MountPath* property. The path of exported data inside the computational task.
- dependentTasks – Tasks dependent on the output have their identifiers listed here.

5.3 TOSCA Emitter

This section describes in detail transformation from editor's class abstraction (section 5.2) to YAML [25] file following the TOSCA standard. While the TOSCA standard was originally created for service deployment, it can also be used in conjunction with Apache Airflow with the help of a custom TOSCA interpreter, as discussed in section 3.3.3.

However, The LEXIS computational workflow is not as abstract as the editor's core. Thus, the workflow cannot be transformed directly and some auxiliary models are required 5.3.7. The design of the structures used in the TOSCA standard emitter is inspired by the LEXIS operators used in Apache Airflow DAG (Direct Acyclic Graph), which represents the pipeline in Apache Airflow. To translate between the TOSCA template and operators, a translator is needed, but its implementation is beyond the scope of this thesis. However, to ensure better usability and validation, the chapter introduces extended TOSCA types for the LEXIS operators in Apache Airflow, which are explained briefly in section 5.3.5.

The emitter from editor's core abstraction to TOSCA is implemented by the visitor design pattern [6]. The visitor design pattern allows us to implement multiple emitters without a need to change the implementation of the editor's core abstraction. Thus, the editor

[6]Design pattern visitor is well described in the book [34]

can support generation of Common Workflow Language[7] or other workflow specification languages on demand. The emitters implements the *IWFEmitter* interface.

For the output of the emitter no tests for validation exists currently. However, the output can be verified by external tool *TOSCA Parser* [22]. The tool was developed by OpenStack community to transform TOSCA deployment specification to HEAT specification (see section 3.3.5).

5.3.1 TOSCA Emitter Implementation

The TOSCA emitter is implemented in the *TOSCAEmitter.cs* file, located within the *WFTOSCAEmitter* namespace. The worfkflow of type *ComputationalWorkflowTemplate* (see section 5.2) is passed as a parameter to the method *Emit*. The instance of class *OrderPreservingDictionary* supplied by the ToscaDOM is created to be the root for the emitted workflow. Then, the *ComposeVersion* and *ComposeMetadata* are called and their output of data type *OrderPair* is added to the root dictionary. This way, TOSCA specification version and workflow metadata are added to the abstract tree composed of the component of the ToscaDOM library. The following code implements addition of the workflow description:

```
if(wfTemplate.Description != default(string))
{
    TOSCATemplate.Add(
        new OrderedPair(
            new StringDataType("description"),
            new StringDataType(wfTemplate.Description)
    ));
}
```

The default path to the LEXIS OASIS TOSCA custom types is added when importing, as described in section 5.3.5. However, additional imports can be specified as well. The OASIS TOSCA v1.2 standard [43] defines that imports can be a file, a repository name, a URL to the file, or a namespace prefix. The node template part consists of the workflow inputs and node templates, as mentioned above. The inputs are composed one by one. The instance of class *CWTTemplateInput* is created from instance of class *CWTemplateInput* or *CWTemplateDataInput* and emitted (the *OrderPair* class instance is created). At the end, computational tasks are composed with overloaded emitter's method *composeTask*, as described in following sections. Nevertheless, the tasks should be ordered by dependencies to ensure, that the dependencies of the properties can be referenced. Therefore, all the nodes are sorted with topological sort algorithm implemented in static method *TopologicalSort* of class *ComputationalWorkflow*. The used algorithm is explained in the book [33] also known as the Kahn's algorithm [28]. However, the AAI session should be kept alive, therefore *LexisAAIOperator* comes before all the tasks and the *LexisAAIStopKeeperOperator* comes after all of them. After the instances of the auxiliary node classes are created, the method of each of the classes can be invoked to add them to the node template section.

[7] CWL is an workflow description language, which aims to enable scientists to share data analysis workflows. More detail about the language can be found in the article [32].

5.3.2 HPC Task

The number of node templates added to the template depends on the *CWHPCTask* input types. When some *CWTaskDataInput* or *CWTaskDataOutput* are present, then the additional operators *HEAppEEnableFileTransferOperator* and *HEAppEEndFileTransferOperator* are generated and property *FileTransferMethodId* is set to *2* in *HEAppEPrepareJobOperator*. Overall, the editor generates the following operators (refer to section 5.3.5) for each of the HPC tasks:

- *HEAppESessionOperator*
- *HEAppEPrepareJobOperator*
- *HEAppESubmitJobOperator*
- *HEAppEEnableFileTransferOperator* – if there is some data inputs or outputs
- *HEAppEEndFileTransferOperator* – if there is some data inputs or outputs
- *HEAppEDeleteJobOperator*
- *HEAppEWaitSubmittedJobSensor*
- *HEAppESessionKeeperSensor*
- *HEAppESessionStopKeeperOperator*

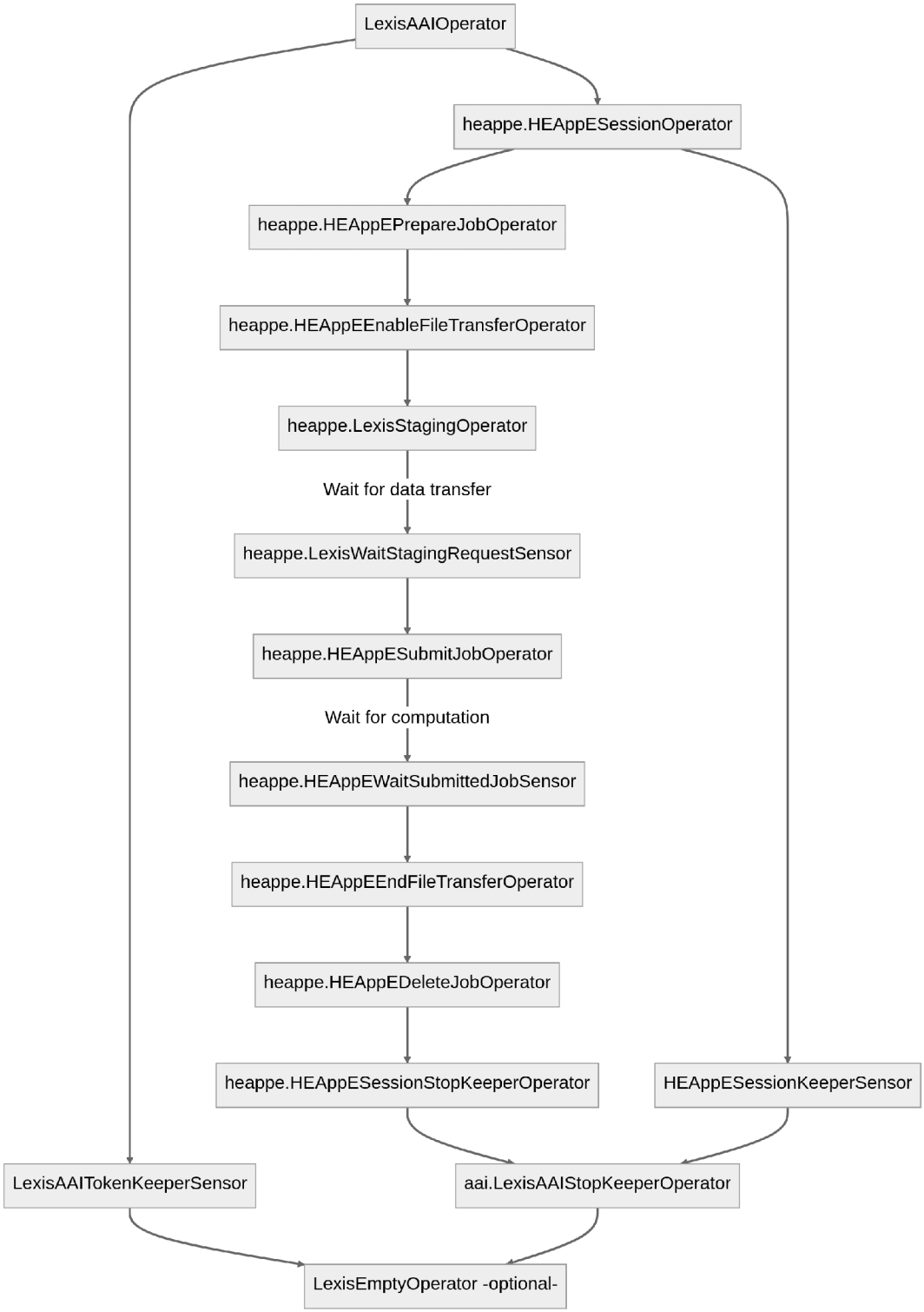


Figure 5.1: TOSCA Emitter Implementation – Example of generated TOSCA nodes for LEXIS

Source: Own diagram

5.3.3 Cloud Task

Cloud task consists only from the one non-optional node, *LexisCloudOperator*.
However, when the data input or output for the task exists, then the data is transferred by *ddi.LexisStagingOperator* to the staging area or from the staging area. The data from the staging area is mounted to the Kubernetes Pod handled by the *LexisCloudOperator*.

5.3.4 Data Transfers

Overall, the data transfers generated by the editor may not be optimal. Nevertheless, the optimization can be done in further development of the editor. Possible scenarios of data transfers are describes in subsections.

Transfer from DDI to Cloud Task

When the cloud task requires some data input from the DDI storage, then the data is fetched from the DDI by *ddi.LexisStagingOperator* and stored to the specific directory for the cloud task. Each workflow has unique secured folder in the staging area. The staged data is mounted to the Kubernetes Pod afterwards.

Transfer from DDI to HPC Task

In case the HPC task requires the data on input, the data is fetched from the DDI by *heappe.LexisStagingOperator* and mounted by SSHFS[8] protocol to the HEAppE's job context directory. The directory management on the staging area is handled automatically by *heappe.LexisStagingOperator*. The transfer has to be enabled before the use of the staging operator.

Transfer between the tasks

To prevent more cloud or HPC tasks from writing to the same source data from the source task, the data is copied for each of the dependent data inputs to their own directory in the staging area. It is not the optimal way, but optimisation of data transfer between tasks is not a topic the thesis is concerned about.

Transfer from Cloud Task to DDI

When the output data from the task is exported to the DDI as a dataset, the directory is created in the staging area and mounted to the cloud Kubernetes Pod. The workflow is designed to transfer the data to the DDI storage after successful execution. However, the current implementation does not account for situations where the input and output data share the same mount point. The operation is handled by the *ddi.LexisStagingOperator*.

Transfer from HPC Task to DDI

The export of the data output to the DDI as a dataset is managed
by *heappe.LexisStagingOperator*. However, the tranfer has to be enabled with *HEAppEEn-*

[8]SSHFS, as the documentation in code repository mention [21], is network filesystem client based on SFTP [35] protocol and FUSE library. FUSE (Filesystem in Userspace [17]) is an interface for userspace programs to export a filesystem to the Linux kernel.

ableFileTransferOperator and proper *FileTransferMethodId* has to be setted in *HEAppEPrepareJobOperator*.

5.3.5 LEXIS Operators Definition for OASIS TOSCA

- *LexisEmptyOperator* – does not execute any action and is used to synchronize the parallel branches of a workflow
- *LexisAAIOperator* – Exchanges the access token given in the workflow input parameter for offline token. The exchange is granted by the AAI authority Keycloak[9].
- *LexisAAITokenKeeperSensor* – Keeps the offline session token active during the workflow execution
- *LexisAAIStopKeeperOperator* – Once the execution reaches the operator, it will stop the *LexisAAITokenKeeperSensor* from keeping the AAI session alive
- *heappe.LexisStagingOperator* – Requests data transfers between staging area, DDI service and HPC cluster. It depends on the *HEAppESessionOperator* and *HEAppEPrepareJobOperator* to have access to the HPC cluster. That is why the operator differs from the *ddi.LexisStagingOperator*.
- *heappe.LexisWaitStagingRequestSensor* – Waits for the result of the data transfer requested by *heappe.LexisStagingOperator*
- *ddi.LexisStagingOperator* – Requests data transfers between staging area and DDI service.
- *ddi.LexisWaitStagingRequestSensor* – Waits for the result of the data transfer requested by *ddi.LexisStagingOperator*
- *HEAppESessionOperator* – Keeps HEAppE session alive.
- *HEAppEPrepareJobOperator* – Prepares the context for the HPC job submission. Sets the properties like target cluster identifier, walltime limit or enables data transfer from staging area together with *HEAppEEnableFileTransferOperator*.
- *HEAppESubmitJobOperator* – Submits a prepared job to the HPC job scheduler queue using the HEAppE middleware
- *HEAppEEnableFileTransferOperator* – Enables data transfer from staging area
- *HEAppEEndFileTransferOperator* – Closes the tunnel for data transfer between staging area and HPC cluster
- *HEAppEDeleteJobOperator* – Removes created context on HPC cluster
- *HEAppEWaitSubmittedJobSensor* – Waits until the job is done

[9]Keycloak is an open source software. It provides identity and access management. It supports protocol OpenID, SAML 2.0 or Kerberos for authentication and Active Directory, LDAP or regular relational database. It is a part of the LEXIS platform as AAI service, as can be seen on figure 3.2. More about the Keycloak can be found on the website [11]

- *HEAppESessionKeeperSensor* – Keeps HEAppE session alive during the computation on HPC cluster
- *HEAppESessionStopKeeperOperator* – When the operator is executed, it stops the *HEAppESessionKeeperSensor* from keeping the HEAppE session alive
- *LexisCloudOperator* – Creates Kubernetes pod[10] on OpenStack, connects volumes with data and observes the execution

5.3.6 OASIS TOSCA

The target TOSCA workflow specification contains the following parts:

- *tosca_definitions_version* – Used TOSCA definition version
- *metadata* – Custom specification metadata. For example, description or author
- *imports* – Import of other TOSCA definition files. The import path for LEXIS Operators Definition for OASIS TOSCA (see section 5.3.5)
- *topology_template*
  - *inputs*
  - *node_templates*

5.3.7 Auxiliary TOSCA Emitter Models

The auxiliary models in the *ComputationalWorkflowTOSCA.cs* file partially abstract the TOSCA structures and help to clarify the implementation. They are complementary to models defined ToscaDOM library used for emitting. All of the models implement interface *IToscaEmitable*. The interface grants implementation of method *Emit* with return value of data type *OrderedPair* defined in ToscaDOM library. ToscaDOM library implements abstract structures representing the TOSCA entities further emitted to the YAML by *YamlDotNet* built-in .NET package. The base class for all structures in the library is *AbstractComponent*. The library is still in development and currently used structures are mostly representation of ordered list with key-value members. Following auxiliary models are defined:

- *CWTRelation* – Represents the relationship between *CWTNode* instances
- *CWTTemplateInput* – Inherits from *CWTemplateInput* core abstract class and implements interface *IToscaEmitable*. Represents the workflow input parameter. It can be instantiated from both *CWTemplateInput* and *CWTemplateDataInput*
- *CWTNodeAttribute* – Represents an attribute of *CWTNode*. It has a name and referenence to the parent *CWTNode*. As specified in the OASIS TOSCA v1.2 standard [43], the attribute definition is:

[10] Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. [13]

> An attribute definition defines a named, typed value that can be associated with an entity defined in this specification (e.g., a Node, Relationship or Capability Type). Specifically, it is used to expose the „actual state“ of some property of a TOSCA entity after it has been deployed and instantiated (as set by the TOSCA orchestrator). Attribute values can be retrieved via the **get_attribute** function from the instance model and used as values to other entities within TOSCA Service Templates.

- *CWTNodePropertyBase* – It is an abstract base class for *CWTNode* property. As stated in the OASIS TOSCA v1.2 standard [43] the property definition is:

  > A property definition defines a named, typed value and related data that can be associated with an entity defined in this specification (e.g., Node Types, Relationship Types, Capability Types, etc.). Properties are used by template authors to provide input values to TOSCA entities which indicate their „desired state“ when they are instantiated. The value of a property can be retrieved using the **get_property** function within TOSCA Service Templates.

  The class itself has properties *Name*, *Required* and reference to the parent *CWTNode*

- *CWTNodePropertyRefAttr* – Inherits from the *CWTNodePropertyBase* and has additional property *AttributeRef* to reference other node's attribute. It also has method *GetAttributeFn* for **get_attribute** construction generation, because the feature of generating **get_attribute** is not yet implemented in ToscaDOM library.

- *CWTNodePropertyRefProp* – Inherits from the *CWTNodePropertyBase* and it has property *PropertyRef* for referencing other node's property. The method *GetPropertyFn* generates **get_property** structure similarly to the method *GetAttributeFn* implemented by *CWTNodePropertyRefAttr*

- *CWTNodePropertyInputRef* – Inherits from the *CWTNodePropertyBase* and it has property *InputRef* for referencing other node's property. The method *GetPropertyFn* generates **get_input** structure similarly to the method *GetInputFn* implemented by *CWTNodePropertyRefProp* and *CWTNodeAttributreRefProp*

- *CWTNodePropertyValue* – Inherits from the *CWTNodePropertyBase* and it directly stores the value of the property and its type.

- *CWTNode* – It is specified in the OASIS TOSCA v1.2 standard [43] as follows:

  > A Node Template specifies the occurrence of a manageable software component as part of an application's topology model which is defined in a TOSCA Service Template. A Node template is an instance of a specified Node Type and can provide customized properties, constraints or operations which override the defaults provided by its Node Type and its implementations.

  The auxiliary class has name, type, reference to the *CWNode* class, list of requirements (list of members with type *CWTRelation*), dictionary of properties *CWTNodePropertyBase*, dictionary of attributes *CWTNodeAttribute* and method for adding previous *CWTNode* to the requirements list

- *CWTNodeAirflow* – is the concrete type of a node defined in custom LEXIS TOSCA types. It additionally has a compulsory attribute called *task_id*
- static class *CWTAAI* – some predefined task names and input names. For example, required workflow input *access_token*
- static class *CWTTypesInterpret* – Interprets values with C#'s *dynamic* data type to the *AbstractComponent* accepted by the ToscaDOM library. The value is distinguished by value type (*CWInputTypes*) passed as the argument

5.4 Graphical Editor Interface

Currently, the minimal valuable prototype of the designed graphical interface introduced in section 4.3 is implemented. That is the workflow editor itself. The user can create a new workflow, add a predefined available computational task, create input dataset, connect data input and output within the workflow nodes and export the workflow to the TOSCA YAML.

Everything except the diagram visualisation with data dependencies between the tasks and workflow input is implemented in .NET Blazor Server framework [40]. The diagram is managed in JavaScript using the JointJS library [10]. Because the changes of the computational workflow state should be handled, the cross invocation of functions and methods is used between the JavaScript and the Blazor Server in C#. Data passed to the function is serialised and deserialised using the converters described in section 5.2.6. The sections below extend the description of JavaScript class abstraction introduced in section 5.2.

5.4.1 Task Diagram

The JavaScript code with implementation is located in file *WorkflowEditor.razor.js*. To highlight that the JavaScript code is for the specific component, the name of the file equals to the name of the file with Blazor web component (*WorkflowEditor.razor*). The *WorkflowEditor* class holds the main context for the diagram and all the nodes and relationships. When the class is initialised, the instances of the classes the diagram *Graph* and *Paper* are created. The *Paper* class represents the view of the *Graph* model according to the JointJS documentation. The *Paper* view can be customized upon initialisation. In our case, the following actions are performed:

- The validation function for the links between the nodes is defined. The validation function checks, if the link can be created between the input and output[11] of the nodes.
- The styling of the view is set
- The tools for link modification are added to the link's view
- *ContextMenu* class is instantiated and added to the event callback. The class defines the context menu design and items. The items included in the context menu depend on the specific context in which it is invoked, which is determined by the type of node in the diagram. Currently, the context menu provides two options: a remove button and a modification button.

[11] In JointJS terminology, the input and output are called ports.

- Zooming behaviour is configured. The scale ratio is propagated to the .NET framework and displayed to the HTML element with predefined identifier
- Drag and drop movement of the whole *Paper* view is initialised. The function responsible for setting the mouse position and a flag that represents a drag operation to the *WorkflowEditor* property *PaperCtx* is registered to the *blank:pointerdown* event. While the user moves with the cursor, function registered to the event *blank:pointermove* moves with the paper with the calculated delta value and sets the cursor position to the *PaperCtx* again. The *blank:pointerup* is considered as the drop action and it ends the dragging setting, the dragging flag to the *false* and unsetting the cursor position in *PaperCtx*

The workflow may have a default state, which can be passed as an argument to the class constructor, that is why the workflow state initialisation happens there. After the initialisation, the nodes are displaced across the *Paper* view with algorithm indicated in the following pseudo code:

```
// the algorithm is in pseudo code

// The nodes are displaced to the grid with column and rows

// nodes without predecessor (also covers workflow's data input nodes)
List<Node> nodesIndegree0 = findIndegree0()

List<Node> visitedNodes = []

// element width constant in pixels,
// algorithm can be improved using the variable width of the nodes
int nodeWidth = 256

// x axis margin constant for the node in pixels
int xDisplacementMargin = 150

// y axis margin constant for the node in pixels
int yDisplacementMargin = 80

// context for counting the offset
// from the heighest row of the displaced nodes
List<List<int>> heightContext = []

// base top offset for the tree of the nodes
// from the nodesIndegree0 list
int treeRowBase = yDisplacementMargin

foreach _ in nodesIndegree0:
    heightContext.push([])
```

```
function placeNode(nodeElement, distance, node0Index):
    // node is not visited yet
    if(nodeElement not in visitedNodes):
        visitedNodes.push(nodeElement.identifier)

        // if the column does not exists in the heightContext
        // then initialise the column
        if(heightContext[node0Index][distance] is undefined):
            heightContext[node0Index][distance] = []

        // sum the current tree root base
        // and all the highest columns of the previous root's trees
        int currentYPosition = treeRowBase
        foreach upperNode of heightContext[node0Index][distance]:
            currentYPostition += upperNode.Height+yDisplacementMargin

        int currentXPosition = xDisplacementMargin
        currentXPosition += distance * ( xDisplacementMargin + nodeWidth )

        nodeElement.setXcoordinate(currentXPosition)
        nodeElement.setYcoordinate(currentYPosition)

        heightContext[node0Index][distance].push(nodeElement.Height)

foreach node0 of nodesIndegree0:
    // the node is not the first one
    int node0Index = nodesIndegree0.findIndex(node0)
    if(node0Index > 0):
        // find the heighest column of the previous tree ,
        // which was placed on the Paper
        int maxHeight = findTheHeighestColumn()
        treeRowBase+=maxHeight

    // Perform BFS from the node0
    // distance -- distance in tree from root node0
    Graph.bfs(
        node0,
        (nodeElement, distance) =>
            placeNode(nodeElement, distance, node0Index)
    )
```

The *WorkflowEditor* also implements following methods:

- *NewDataInput* – Creates a new instance of the *TemplateDataInput* with given name and identifier. Adds it to the workflow and diagram using the method *AddTemplateDataInput* of the *WorkflowEditor* class
- *SetDataInputName* – Sets name of the data input with specific identifier. It is triggered, when the name changed in the workflow input's menu (see section 5.4.2)
- *RemoveTemplateDataInput* – Removes the data input node with given identifier from the diagram and workflow
- *SetTaskName* – Sets name of the task with specific identifier. The task name can be changed from task menu (see section 5.4.4)
- *EnableDatasetOutputExport* and *DisableDatasetOutputExport* – Enables or disables export of the computational task output. It changes the point's colour of the target output between the red and purple colour
- *NewNode* – Creates a new instance of the *Node* class from the serialised *CWTask* class and adds it to the diagram
- *RemoveNode* – Removes a node with given identifier

The nodes of the diagram are styled in class *Node*. They are styled with attributes, selectors and properties implemented by the JointJS library. The inputs and outputs are placed on the sides of the node. The node's height depends on their number. When the output or input is added or removed, new height of the node is computed with method *ComputeNewSize*.

5.4.2 Computation Workflow Inputs Menu

The workflow input menu is accessible from the right navigation bar (see figures D.1 and D.5). The menu contains a list of workflow inputs. Some items are restricted, hence they cannot be deleted or renamed, it is possible otherwise. The unrestricted inputs can be marked as required and the name or default value can be modified. There are two buttons for addition of the inputs on the bottom of the list. The first is for the data input addition, which also triggers *NewDataInput* method of the class *WorkflowEditor* in JavaScript and the second is for the basic value-based input.

5.4.3 Addition of the Computational Task

To add a computational task to the workflow, user can click on the button with the plus icon in the right navigation menu. Then, the modal window offers a select box with a list of available computational tasks. The *HPCTaskService* and *CloudTaskService* collect the available computational tasks from the HEAppE and from the Cloud Image Metadata Registry (see section 4.2). Currently, the available tasks are mocked statically.

5.4.4 Task Menu

The task menu appears, when user clicks on the edit item in context menu (see figure D.7). The modal window lists the inputs and outputs of the computational task and user can change their default values (see figure D.8). In case of the data output, the user can click to the switch to export it. After that, the colour representation of the output in the diagram is changed.

5.4.5 Exporting the workflow to the TOSCA YAML file

When the user wants to export the composed computational workflow, he clicks on the button with download icon next to the title. The button triggers the workflow emitter (described in section 5.3) for the TOSCA format. When the TOSCA workflow specification is prepared, then the JavaScript function *DownloadFile* is invoked and the specification is passed as the string data type in argument. The function creates the *Blob*[12] object and it downloads the file to the user's computer via the browser (see figure D.12).

5.4.6 User's Feedback

A prototype graphical editor was tested by 4 potential users who were given specific tasks to complete. The feedback from these users will be used to improve the editor's interface and make it more user-friendly and achieve a better user experience. The feedback group consisted of two types of users: those with in-depth knowledge of composing computational workflows using the LEXIS system, and those who use LEXIS to execute pre-designed workflows. Users were given the following tasks:

- Add a computational task to the workflow
- Add workflow's data input and give it a name and connect it to a computational task
- Remove the created workflow's data input
- Export the data output from the computational task
- Download the workflow's specification to your computer

Based on user feedback, the data input in the workflow should be separated or distinguished more clearly from other inputs in the menu. In addition, the users advised folding items in the menu to achieve a compact view. Then, the users can expand only relevant items to them. Some users expressed confusion regarding the icons used in the navigation menu for accessing the input and output menus. Nevertheless, they appreciated the visualisation and simplicity of the task on the diagram. They would appreciate further improvements, such as a description of the connections between the data outputs and inputs of computational tasks, as well as quick access to the workflow input menu when a user clicks on a workflow data input on the diagram. An additional improvement that would be welcomed is to include a node representation on the diagram for the exported dataset.

[12]The *Blob* object represents a blob, which is a file-like object of immutable, raw data. Cited from the specification [9]

Chapter 6

Conclusion

The thesis is closely related to the LEXIS platform, and many decisions and technologies used depend on the platform's other components. At the outset, I engaged in discussions with members of the LEXIS team and HPC cluster users regarding the concept of a workflow editor. The initial ideas were varied, but after several iterations, a consensus was reached to develop an editor that was provided to both computational workflow designers and users without extensive knowledge of computer science. The primary objective of the editor was to facilitate a more user-friendly approach to composing workflows for multi-location and multi-architecture computing. After creating the non-functional graphical prototype, I presented it to the involved people, gathered their feedback, which I have then discussed with them. Based on their input, I proceeded to design the non-interface components of the editor, which relied on the Apache Airflow workflow management platform and TOSCA specification. There was no existing way to translate TOSCA into Apache Airflow, and we also considered the possibility of using a non-standardised specification instead of the TOSCA standard for workflow specification. However, we found the TOSCA standard more suitable for our use case. Therefore, I developed a translator prototype that converts TOSCA into Python classes that are specific to the Apache Airflow platform. Once I became familiar with the process of composing workflows, I began designing the abstractions for the editor. Once the design was finished, I started the implementation of the editor's core, the workflow abstraction and TOSCA emitter. I tested the possibility of composing the computational workflow before I started to develop the graphical user interface. In my opinion, the most challenging part of the thesis was implementing the TOSCA emitter and editor abstraction.

In conclusion, the graphical editor is a valuable tool for streamlining the research process, not only for scientists. Its graphical interface and useful features make it an ideal solution for researchers and others who work with advanced computational workflows and who want to improve their efficiency and productivity. With the help of editor and the LEXIS platform, the researchers can focus on what matters – their research.

The graphical editor is built on the .NET Blazor Server framework with a JointJS library for diagram visualisation, it ensures the extensibility of the features, the other emitters of the workflow specification can be implemented, and it suggests a graphical interface. The graphical editor prototype is ready for further development and usage in the European project EXA4MIND [20] and the LEXIS platform. The LEXIS platform is still in active development and as a result, some of the action points mentioned earlier have been postponed until completion at a later date.

The editor is expected to undergo further extensions beyond what is outlined in the following points. In the nearest future, the development of the application will continue by completing the following action points.

- Validation tests for the output of the emitter (see section 5.3)
- End-to-end editor testing
- Integration of the editor to the LEXIS platform
- Implementation of remaining parts of the GUI (see section 4.3)

Bibliography

[1] *About the Loschmidt Laboratories* [online]. Loschmidt Laboratories [cit. 2023-01-18]. Available at: https://loschmidt.chemi.muni.cz/.

[2] *Apache Airflow* [online]. [cit. 2023-04-27]. Available at: https://airflow.apache.org/.

[3] *Blazor.Diagrams – Diagrams library for the Blazor framework* [online]. [cit. 2023-04-30]. Available at: https://github.com/Blazor-Diagrams/Blazor.Diagrams.

[4] *FireProt-ASR tool* [online]. Loschmidt Laboratories [cit. 2023-01-18]. Available at: https://loschmidt.chemi.muni.cz/fireprotasr/?action=help.

[5] *HEAppE Yorc plugin* [online]. LEXIS project [cit. 2023-01-18]. Available at: https://github.com/lexis-project/orch-service-yorc-heappe-plugin.

[6] *Heat Orchestration Template (HOT) specification* [online]. OpenStack Foundation [cit. 2023-04-17]. Available at: https://docs.openstack.org/heat/latest/template_guide/hot_spec.html.

[7] *Hybrid cloud/HPC TOSCA orchestrator* [online]. GitHub [cit. 2023-05-01]. Available at: https://github.com/ystia/yorc.

[8] *IRODS - The Integrated Rule-Oriented Data System* [online]. iRODS Consortium [cit. 2023-01-18]. Available at: https://irods.org/.

[9] *JavaScript – Web API Reference – Blob object* [online]. [cit. 2023-04-30]. Available at: https://developer.mozilla.org/en-US/docs/Web/API/Blob.

[10] *JavaScript diagramming library for interactive UIs – JointJS* [online]. [cit. 2023-04-30]. Available at: https://www.jointjs.com/.

[11] *Keycloak - Open Source Identity and Access Management* [online]. [cit. 2023-04-28]. Available at: https://www.keycloak.org/.

[12] *Kubernetes* [online]. [cit. 2023-04-28]. Available at: https://kubernetes.io.

[13] *Kubernetes - Pod* [online]. [cit. 2023-04-28]. Available at: https://kubernetes.io/docs/concepts/workloads/pods/.

[14] *LEXIS - Large-scale EXecution for Industry & Society* [online]. LEXIS Project Consortium [cit. 2023-01-18]. Available at: https://lexis-project.eu/web/.

[15] *LEXIS objectives* [online]. LEXIS project [cit. 2022-12-29]. Available at: https://lexis-project.eu/web/high-performance-computing/objectives-2/.

[16] *The LEXIS project public documentation* [online]. The LEXIS Platform Team [cit. 2023-04-27]. Available at: https://docs.lexis.tech/.

[17] *Linux FUSE Library* [online]. [cit. 2023-04-29]. Available at: https://github.com/libfuse/libfuse/.

[18] *OICD Yorc plugin* [online]. LEXIS project [cit. 2023-01-18]. Available at: https://github.com/lexis-project/orch-service-yorc-oidc-client.

[19] *OpenID - Specification* [online]. OpenID Foundation [cit. 2023-01-18]. Available at: https://openid.net/developers/specs/.

[20] *The Platform for Extreme Data* [online]. [cit. 2023-04-30]. Available at: https://exa4mind.eu/.

[21] *SSHFS – Network filesystem client* [online]. [cit. 2023-04-29]. Available at: https://github.com/libfuse/sshfs/.

[22] *TOSCA Parser* [online]. [cit. 2023-04-30]. Available at: https://pypi.org/project/tosca-parser/.

[23] *What is DevOps?* [online]. Microsoft [cit. 2023-01-18]. Available at: https://learn.microsoft.com/en-us/devops/what-is-devops.

[24] *What is HPC?* [online]. IBM [cit. 2022-12-29]. Available at: https://www.ibm.com/topics/hpc.

[25] *YAML - YAML Ain't Markup Language™* [online]. [cit. 2023-01-18]. Available at: https://yaml.org/.

[26] *Yorc DDI plugin* [online]. LEXIS project [cit. 2023-01-18]. Available at: https://github.com/lexis-project/orch-service-yorc-ddi-plugin.

[27] *Ystia Suite* [online]. Atos SE [cit. 2023-01-18]. Available at: https://ystia.github.io/.

[28] *Graph Topological Sort — Kahn's Algorithm* [online]. 2021 [cit. 2023-04-29]. Available at: https://adelachao.medium.com/graph-topological-sort-kahns-algorithm-93380b00e7d7.

[29] Banks, A. and Porcello, E. *Learning React: Modern Patterns for Developing React Apps*. O'Reilly Media, 2020. ISBN 9781492051695. Available at: https://books.google.cz/books?id=tDjrDwAAQBAJ.

[30] Bharathi, S., Chervenak, A., Deelman, E., Mehta, G., Su, M.-H. et al. Characterization of Scientific Workflows. [online]. DOI: 10.1109/WORKS.2008.4723958. ISSN 2151-1381. Available at: https://doi.org/10.1109/WORKS.2008.4723958.

[31] Challita, S. *TOSCA Studio* [online]. [cit. 2023-01-18]. Available at: https://github.com/occiware/TOSCA-Studio/.

[32] Crusoe, M. R., Abeln, S., Iosup, A., Amstutz, P., Chilton, J. et al. Methods Included: Standardizing Computational Reuse and Portability with the Common Workflow Language. The CWL Community Communications of the ACM. 2022, Vol. 65 No. 6, p. 54–63. DOI: 10.1145/3486897.

[33] Demel, J. Grafy a jejich aplikace. In:. Academia, 2002, chap. 5.2.7. ISBN 8020009906.

[34] Freeman, E. and Robson, E. *Head First Design Patterns*. O'Reilly Media, 2020. ISBN 9781492077978. Available at: https://books.google.cz/books?id=Lw8LEAAAQBAJ.

[35] Galbraith, J. and Saarenmaa, O. *SSH File Transfer Protocol*. Internet-Draft draft-ietf-secsh-filexfer-13. Internet Engineering Task Force, july 2006. Work in Progress. Available at: https://datatracker.ietf.org/doc/draft-ietf-secsh-filexfer/13/.

[36] Ganne, L. Design and Implementation of the HPC-Federated Orchestration System - Intermediate. Bull/Atos. 2020, [cit. 2023-01-20]. DOI: 10.3030/825532. v1.1. Available at: https://ec.europa.eu/research/participants/documents/downloadPublic?documentIds=080166e5cca47e99&appId=PPGMS.

[37] Goubier, T., Rakowsky, N. and Harig, S. Fast Tsunami Simulations for a Real-Time Emergency Response Flow. In: *2020 IEEE/ACM HPC for Urgent Decision Making (UrgentHPC)*. 2020, p. 21–26. DOI: 10.1109/UrgentHPC51945.2020.00008.

[38] Heineman, G. *Learning Algorithms*. O'Reilly Media, 2021. ISBN 9781492091011. Available at: https://learning.oreilly.com/library/view/learning-algorithms/9781492091059/.

[39] Leach, P., Mealling, M. and Salz, R. *A Universally Unique Identifier (UUID) Urn Namespace Specification*. RFC 4122. Internet Engineering Task Force, July 2005. Available at: https://tools.ietf.org/html/rfc4122.

[40] Litvinavicius, T. *Exploring Blazor: Creating Server-side and Client-side Applications in .NET 7*. Apress, 2022. ISBN 9781484287675. Available at: https://books.google.cz/books?id=Bc5OzwEACAAJ.

[41] Ludäscher, B., Bowers, S. and McPhillips, T. Scientific Workflows. In: LIU, L. and ÖZSU, M. T., ed. *Encyclopedia of Database Systems*. Boston, MA: Springer US, 2009, p. 2507–2511. DOI: 10.1007/978-0-387-39940-9_1471. ISBN 978-0-387-39940-9. Available at: https://doi.org/10.1007/978-0-387-39940-9_1471.

[42] Magarielli, D. Avio Aero use cases: review of Aeronautics use cases' KPIs and expected impact on Aeronautical market. 2021, [cit. 2023-01-18]. DOI: 10.3030/825532. v1.1. Available at: https://ec.europa.eu/research/participants/documents/downloadPublic?documentIds=080166e5e644b2ad&appId=PPGMS.

[43] OASIS. *OASIS TOSCA standard v1.2* [online]. 2019 [cit. 2023-04-29]. Available at: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/os/TOSCA-Simple-Profile-YAML-v1.2-os.pdf.

[44] Swiatkowski, J. Slurm on OpenStack Computing Platform for Development. [online]. p. 2, [cit. 2023-01-18]. Available at: https://raw.githubusercontent.com/jsw0011/slurm-openstack-devel-plugin/main/2022-PPFIT-SlurmOpenStackPluginDevel.pdf.

[45] T. Bray, E. *The JavaScript Object Notation (JSON) Data Interchange Format* [online]. RFC 8259. 2017 [cit. 2023-04-25]. Available at: https://www.rfc-editor.org/info/std90.

[46] Wilkinson, B. Grid Computing: Techniques and Applications. In:. Taylor & Francis, 2009, chap. 8.2.1. Chapman & Hall/CRC Computational Science. ISBN 9781420069532. Available at: https://learning.oreilly.com/library/view/grid-computing/9781420069549/.

Appendix A

LEXIS Operator's Types in OASIS TOSCA

TOSCA specification for extended node types and groups that are specific to the LEXIS platform.

```
tosca_definitions_version: tosca_simple_yaml_1_2


metadata:
  author: IT4Innovations
  version: 1.2.0
  name: lexis.airflow.types

description: LEXIS Airflow TOSCA types for workflow execution

# LEXIS DATA TYPES

data_types:
  lexis.datatypes.ddi.Metadata:
    derived_from: tosca.datatypes.Root
    properties:
      creator:
        description: Dataset creators
        type: list
        entry_schema:
          type: string
        required: False
      contributor:
        description: Dataset contributors
        type: list
        entry_schema:
          type: string
        required: False
      publisher:
        description: Dataset publishers
```

```
      type: list
      entry_schema:
        type: string
      required: False
    owner:
      description: Dataset owners
      type: list
      entry_schema:
        type: string
      required: False
    internalID:
      description: Dataset identifier
      type: string
      required: False
    publicationYear:
      description: Dataset year of publication
      type: string
      required: False
    resourceType:
      description: Dataset resource type
      type: string
      required: False
    title:
      description: Dataset title
      type: string
      required: False
  lexis.datatypes.ddi.templateinput:
    derived_from: tosca.datatypes.Root
    properties:
      mount_point:
        type: string
        description: relative path to the root of the HPC job
        required: True
      ddi_path:
        type: string
        required: False
  lexis.datatypes.cloud.mountpoint:
    derived_from: tosca.datatypes.Root
    properties:
      mount_point:
        type: string
        required: True
      staging_operator_task_id:
        type: string
        required: True


# LEXIS Airflow nodes types
```

```yaml
node_types:
  # BASE
  lexis.nodes.operators.base.LexisOperatorBase:
    derived_from: tosca.nodes.Root
    attributes:
      task_id:
        type: string
    capabilities:
      basicTask:
        type: lexis.capabilities.basicTask
  # Empty operator
  lexis.nodes.operators.base.LexisEmptyOperator:
    derived_from: tosca.nodes.Root
    attributes:
      task_id:
        type: string
    capabilities:
      basicTask:
        type: lexis.capabilities.basicTask
    requirements:
      - previous_task:
        node: lexis.nodes.operators.base.LexisOperatorBase
        relationship: tosca.relationships.DependsOn
        occurrences: [1, UNBOUNDED]
  # AAI Operators
  lexis.nodes.operators.aai.LexisAAIOperator:
    derived_from: tosca.nodes.Root
    description: Exchange of access token for the offline token
    to access LEXIS services
    properties:
      access_token:
        type: string
    attributes:
      task_id:
        type: string
  lexis.nodes.operators.aai.LexisAAITokenKeeperSensor:
    derived_from: lexis.nodes.operators.base.LexisOperatorBase
    description: Keep refreshing the token for accessing LEXIS services
    attributes:
      task_id:
        type: string
    requirements:
      - aai_operator:
          node: lexis.nodes.operators.aai.LexisAAIOperator
          relationship: tosca.relationships.DependsOn
          occurrences: [1, UNBOUNDED]
  lexis.nodes.operators.aai.LexisAAIStopKeeperOperator:
```

```
  derived_from: lexis.nodes.operators.base.LexisOperatorBase
  description: Stop LexisAAITokenKeeperSensor when computation ends
  attributes:
    task_id:
      type: string
  requirements:
    - previous_task:
        node: lexis.nodes.operators.base.LexisOperatorBase
        relationship: tosca.relationships.DependsOn
        occurrences: [1, UNBOUNDED]
# DDI Operators
lexis.nodes.operators.heappe.LexisStagingOperator:
  derived_from: lexis.nodes.operators.base.LexisOperatorBase
  properties:
    source_system:
      type: string
      required: True
    source_path:
      type: string
      required: True
    target_system:
      type: string
      required: True
    target_path:
      type: string
      required: True
    heappe_prepare_job_task_id:
      type: string
      required: True
    heappe_enable_file_transfer_task_id:
      type: string
      required: True
    encryption:
      type: boolean
      required: False
      default: False
    compression:
      type: boolean
      required: False
      default: False
    metadata:
      type: lexis.datatypes.ddi.Metadata
      required: False
      default: null
  attributes:
    task_id:
      type: string
  requirements:
```

```
    - heappe_prepare_job_task_id:
        node: lexis.nodes.operators.heappe.HEAppEPrepareJobOperator
        relationship: lexis.relationships.heappe.HEAppEPrepareJob
        occurrences: [1, 1]
    - heappe_enable_file_transfer_task_id:
        node: lexis.nodes.operators.heappe.HEAppEEnableFileTransferOperator
        relationship: lexis.relationships.heappe.HEAppEFileTransferEnableTask
        occurrences: [1, 1]
    - previous_task:
        node: lexis.nodes.operators.base.LexisOperatorBase
        relationship: tosca.relationships.DependsOn
        occurrences: [1, UNBOUNDED]
lexis.nodes.operators.heappe.LexisWaitStagingRequestSensor:
  derived_from: lexis.nodes.operators.base.LexisOperatorBase
  description: Waits for staging operation
  properties:
    staging_operator_task_id:
      type: string
      required: True
  attributes:
    task_id:
      type: string
  requirements:
    - staging_operator_task_id:
        node: lexis.nodes.operators.heappe.LexisStagingOperator
        relationship: lexis.relationships.ddi.DDITask
        occurrences: [1, 1]
lexis.nodes.operators.ddi.LexisStagingOperator:
  derived_from: lexis.nodes.operators.base.LexisOperatorBase
  properties:
    source_system:
      type: string
      required: True
    source_path:
      type: string
      required: True
    target_system:
      type: string
      required: True
    target_path:
      type: string
      required: True
    encryption:
      type: boolean
      required: False
      default: False
    compression:
      type: boolean
```

```
      required: False
      default: False
    metadata:
      type: lexis.datatypes.ddi.Metadata
      required: False
      default: null
  attributes:
    task_id:
      type: string
  requirements:
    - previous_task:
        node: lexis.nodes.operators.base.LexisOperatorBase
        relationship: tosca.relationships.DependsOn
        occurrences: [1, UNBOUNDED]
lexis.nodes.operators.ddi.LexisWaitStagingRequestSensor:
  derived_from: lexis.nodes.operators.base.LexisOperatorBase
  description: Waits for staging operation
  properties:
    staging_operator_task_id:
      type: string
      required: True
  attributes:
    task_id:
      type: string
  requirements:
    - staging_operator_task_id:
        node: lexis.nodes.operators.ddi.LexisStagingOperator
        relationship: lexis.relationships.ddi.DDITask
        occurrences: [1, 1]

# HEAppE Operators
lexis.nodes.operators.heappe.HEAppESessionOperator:
  derived_from: lexis.nodes.operators.base.LexisOperatorBase
  properties:
    heappe_uri:
      type: string
      required: True
  attributes:
    task_id:
      type: string
  requirements:
    - aai_operator:
        node: lexis.nodes.operators.aai.LexisAAIOperator
    - previous_task:
        node: lexis.nodes.operators.base.LexisOperatorBase
        relationship: tosca.relationships.DependsOn
        occurrences: [1, UNBOUNDED]
lexis.nodes.operators.heappe.HEAppEPrepareJobOperator:
```

```
derived_from: lexis.nodes.operators.base.LexisOperatorBase
properties:
  Name:
    type: string
    required: True
  Project:
    type: string
    required: True
  ClusterId:
    type: number
    required: True
  CommandTemplateId:
    type: number
    required: True
  FileTransferMethodId:
    type: number
    # TODO: description about methods!!!
    required: True
  EnvironmentVariables:
    type: map
    required: True
  MinCores:
    type: number
    required: True
    default: 1
  MaxCores:
    type: number
    required: True
  WalltimeLimit:
    type: number
    required: True
    default: 600
  Priority:
    type: number
    required: True
  ClusterNodeTypeId:
    type: number
    required: True
  TemplateParameterValues:
    type: map
    required: True
  heappe_session_task_id:
    type: string
    required: True
attributes:
  task_id:
    type: string
requirements:
```

```
    - heappe_session_task_id:
        node: lexis.nodes.operators.heappe.HEAppESessionOperator
        relationship: lexis.relationships.heappe.HEAppESession
        occurrences: [1, 1]
lexis.nodes.operators.heappe.HEAppESubmitJobOperator:
  derived_from: lexis.nodes.operators.base.LexisOperatorBase
  properties:
    heappe_session_task_id:
      type: string
      required: True
    heappe_prepare_job_task_id:
      type: string
      required: True
  attributes:
    task_id:
      type: string
  requirements:
    - heappe_session_task_id:
        node: lexis.nodes.operators.heappe.HEAppESessionOperator
        relationship: lexis.relationships.heappe.HEAppESession
        occurrences: [1, 1]
    - heappe_prepare_job_task_id:
        node: lexis.nodes.operators.heappe.HEAppEPrepareJobOperator
        relationship: lexis.relationships.heappe.HEAppEPrepareJob
        occurrences: [1, 1]
    - previous_task:
        node: lexis.nodes.operators.base.LexisOperatorBase
        relationship: tosca.relationships.DependsOn
        occurrences: [1, UNBOUNDED]
lexis.nodes.operators.heappe.HEAppEEnableFileTransferOperator:
  derived_from: lexis.nodes.operators.base.LexisOperatorBase
  properties:
    heappe_session_task_id:
      type: string
      required: True
    heappe_prepare_job_task_id:
      type: string
      required: True
  attributes:
    task_id:
      type: string
  requirements:
    - heappe_session_task_id:
        node: lexis.nodes.operators.heappe.HEAppESessionOperator
        relationship: lexis.relationships.heappe.HEAppESession
        occurrences: [1, 1]
    - heappe_prepare_job_task_id:
        node: lexis.nodes.operators.heappe.HEAppEPrepareJobOperator
```

```
        relationship: lexis.relationships.heappe.HEAppEPrepareJob
        occurrences: [1, 1]
lexis.nodes.operators.heappe.HEAppEEndFileTransferOperator:
  derived_from: lexis.nodes.operators.base.LexisOperatorBase
  properties:
    heappe_session_task_id:
      type: string
      required: True
    heappe_prepare_job_task_id:
      type: string
      required: True
    heappe_enable_file_transfer_task_id:
      type: string
      required: True
  attributes:
    task_id:
      type: string
  requirements:
    - heappe_session_task_id:
        node: lexis.nodes.operators.heappe.HEAppESessionOperator
        relationship: lexis.relationships.heappe.HEAppESession
        occurrences: [1, 1]
    - heappe_prepare_job_task_id:
        node: lexis.nodes.operators.heappe.HEAppEPrepareJobOperator
        relationship: lexis.relationships.heappe.HEAppEPrepareJob
        occurrences: [1, 1]
    - heappe_enable_file_transfer_task_id:
        node: lexis.nodes.operators.heappe.HEAppEEnableFileTransferOperator
        relationship: lexis.relationships.heappe.HEAppEFileTransferEnableTask
        occurrences: [1, 1]
    - previous_task:
        node: lexis.nodes.operators.base.LexisOperatorBase
        relationship: tosca.relationships.DependsOn
        occurrences: [1, UNBOUNDED]
lexis.nodes.operators.heappe.HEAppEDeleteJobOperator:
  derived_from: lexis.nodes.operators.base.LexisOperatorBase
  properties:
    heappe_session_task_id:
      type: string
      required: True
    heappe_prepare_job_task_id:
      type: string
      required: True
  attributes:
    task_id:
      type: string
  requirements:
    - heappe_session_task_id:
```

```
        node: lexis.nodes.operators.heappe.HEAppESessionOperator
        relationship: lexis.relationships.heappe.HEAppESession
        occurrences: [1, 1]
    - heappe_prepare_job_task_id:
        node: lexis.nodes.operators.heappe.HEAppEPrepareJobOperator
        relationship: lexis.relationships.heappe.HEAppEPrepareJob
        occurrences: [1, 1]
    - previous_task:
        node: lexis.nodes.operators.base.LexisOperatorBase
        relationship: tosca.relationships.DependsOn
        occurrences: [1, UNBOUNDED]
lexis.nodes.operators.heappe.HEAppEWaitSubmittedJobSensor:
  derived_from: lexis.nodes.operators.base.LexisOperatorBase
  properties:
    heappe_session_task_id:
      type: string
      required: True
    heappe_submitted_job_task_id:
      type: string
      required: True
  attributes:
    task_id:
      type: string
  requirements:
    - heappe_session_task_id:
        node: lexis.nodes.operators.heappe.HEAppESessionOperator
        relationship: lexis.relationships.heappe.HEAppESession
        occurrences: [1, 1]
    - heappe_submitted_job_task_id:
        node: lexis.nodes.operators.heappe.HEAppESubmitJobOperator
        relationship: lexis.relationships.heappe.HEAppESubmittedJob
        occurrences: [1, 1]
lexis.nodes.operators.heappe.HEAppESessionKeeperSensor:
  derived_from: lexis.nodes.operators.base.LexisOperatorBase
  properties:
    heappe_session_task_id:
      type: string
      required: True
    heappe_session_stop_task_id:
      type: string
      required: True
  attributes:
    task_id:
      type: string
  requirements:
    - heappe_session_task_id:
        node: lexis.nodes.operators.heappe.HEAppESessionOperator
        relationship: lexis.relationships.heappe.HEAppESession
```

```
        occurrences: [1, 1]

lexis.nodes.operators.heappe.HEAppESessionStopKeeperOperator:
  derived_from: lexis.nodes.operators.base.LexisOperatorBase
  description: Puts condition into Airflow's XCOM to stop HEAppESessionKeeperSensor f
  attributes:
    task_id:
      type: string
  requirements:
    - previous_task:
        node: lexis.nodes.operators.base.LexisOperatorBase
        relationship: tosca.relationships.DependsOn
        occurrences: [1, UNBOUNDED]
# Cloud Operators
lexis.nodes.operators.cloud.LexisCloudOperator:
  derived_from: lexis.nodes.operators.base.LexisOperatorBase
  description: Cloud job task
  properties:
    name:
      type: string
      required: True
    image:
      type: string
      required: True
    labels:
      type: map
      required: False
    env_vars:
      type: map
      required: False
    dataset_input:
      type: list
      description: Mount points for input datasets
      entry_schema:
        type: lexis.datatypes.cloud.mountpoint
      required: False
    dataset_output:
      type: list
      description: Mount points for output datasets
      entry_schema:
        type: lexis.datatypes.cloud.mountpoint
      required: False
  attributes:
    task_id:
      type: string
  requirements:
    - previous_task:
        node: lexis.nodes.operators.base.LexisOperatorBase
```

```
          relationship: tosca.relationships.DependsOn
          occurrences: [1, UNBOUNDED]


# Groups
group_types:
  lexis.groups.heappe.HPCLocationGroup:
    derived_from: tosca.groups.Root
    description: Run HPC job on specific location
    properties:
      heappe_uri:
        type: string

# Policies

policy_types:
  lexis.policies.HPCLocation:
    derived_from: tosca.policies.Root
    description: HPC location policy. When applied,
    targets with HPC job will get specified location (HEAppE URI).
    properties:
      heappe_uri:
        type: string
    targets:
      - lexis.groups.heappe.HPCLocationGroup
  lexis.policies.workflowRepeat:
    description: Repeat the part of the workflow X times.
    properties:
      number_of_repetition:
          type: number
          required: True
    targets:
      - tosca.groups.Root


# Capabilities
capability_types:
  lexis.capabilities.basicTask:
    derived_from: tosca.capabilities.Root
    description: >
      Basic capability of each Airflow task
    attributes:
      task_id:
        type: string
        description: Airflow task ID
  # HEAppE
  lexis.capabilities.heappe.HEAppESessionProp:
    derived_from: lexis.capabilities.basicTask
```

```
    description: >
      Requires HEAppE session task ID
    properties:
      heappe_session_task_id:
        type: string
        required: True
  lexis.capabilities.heappe.HEAppESessionKeeperStopProp:
    derived_from: lexis.capabilities.basicTask
    description: >
      Requires HEAppE stop session task ID
    properties:
      heappe_session_stop_task_id:
        type: string
        required: True
  lexis.capabilities.heappe.HEAppEFileTransferEnableProp:
    derived_from: lexis.capabilities.basicTask
    description: >
      Requires HEAppE file tranfer enable task ID
    properties:
      heappe_enable_file_transfer_task_id:
        type: string
        required: True
  lexis.capabilities.heappe.HEAppEPrepareJobProp:
    derived_from: lexis.capabilities.basicTask
    description: >
      Requires HEAppE prepare job node task ID
    properties:
      heappe_prepare_job_task_id:
        type: string
        required: True
  lexis.capabilities.heappe.HEAppESubmittedProp:
    derived_from: lexis.capabilities.basicTask
    description: >
      Requires HEAppE submitted job node task ID
    properties:
      heappe_submitted_job_task_id:
        type: string
        required: True
  # DDI
  lexis.capabilities.ddi.LEXISStagingOperator:
    derived_from: lexis.capabilities.basicTask
    description: >
      Requires LEXIS DDI staging node task ID
    properties:
      staging_operator_task_id:
        type: string
        required: True
# Relationships
```

```yaml
relationship_types:
  lexis.relationships.aai.AAIToken:
    derived_from: tosca.relationships.DependsOn
    description: Requires AAI token.
  # HEAppE related relationships
  lexis.relationships.heappe.HEAppESession:
    derived_from: tosca.relationships.DependsOn
    description: Dependency on HEAppE session task

    valid_target_types:
      - lexis.capabilities.heappe.HEAppESessionProp
  lexis.relationships.heappe.HEAppESessionKeeperStop:
    derived_from: tosca.relationships.DependsOn
    description: Dependency on HEAppE session stop task.

    valid_target_types:
      - lexis.capabilities.heappe.HEAppESessionKeeperStopProp
  lexis.relationships.heappe.HEAppESubmittedJob:
    derived_from: tosca.relationships.DependsOn
    description: Dependency on HEAppE submitted job task

    valid_target_types:
      - lexis.capabilities.heappe.HEAppESubmittedProp
  lexis.relationships.heappe.HEAppEPrepareJob:
    derived_from: tosca.relationships.DependsOn
    description: Dependency on HEAppE session task

    valid_target_types:
      - lexis.capabilities.heappe.HEAppEPrepareJobProp
  lexis.relationships.heappe.HEAppEFileTransferEnableTask:
    derived_from: tosca.relationships.DependsOn
    description: Dependency on HEAppE file transfer context

    valid_target_types:
      - lexis.capabilities.heappe.HEAppEFileTransferEnableProp
  # DDI related relationships
  lexis.relationships.ddi.DDITask:
    derived_from: tosca.relationships.DependsOn
    description: Dependency on dataset transfer task

    valid_target_types:
      - lexis.capabilities.basicTask
      - lexis.capabilities.ddi.LEXISStagingOperator
```

Appendix B

Example of generated TOSCA

Example of simple HPC computational workflow exported from the editor to the TOSCA specification.

```yaml
tosca_definitions_version: tosca_simple_yaml_1_2
metadata:
  template_name: MyFirstWorkflow
  template_version: v1_0
  template_author: Singularita
description: The internet is in your pocket
imports:
- file:../lexis-airflow-types.yml
topology_template:
  inputs:
    access_token:
      required: true
      type: string
    precision:
      required: false
      type: integer
      default: 6
    computation_project:
      required: false
      type: string
    max_cores:
      required: false
      type: integer
  node_templates:
    aai_token_init:
      type: lexis.nodes.operators.aai.LexisAAIOperator
      properties:
        access_token: { get_input: access_token }
    1_heappe_session:
      type: lexis.nodes.operators.heappe.HEAppESessionOperator
      properties:
        heappe_uri: https://heappe.it4i.cz
```

```
  requirements:
  - aai_operator: aai_token_init
  - previous_task: aai_token_init
1_heappe_prepare_job:
  type: lexis.nodes.operators.heappe.HEAppEPrepareJobOperator
  properties:
    heappe_session_task_id: {
        get_attribute: [ 1_heappe_session,task_id ]
    }
    Name: HPCTaskOne
    Project: { get_input: computation_project }
    ClusterId: 0
    CommandTemplateId: 3
    MinCores: 1
    MaxCores: { get_input: max_cores }
    WalltimeLimit: 128
    Priority: 4
    ClusterNodeTypeId: 1
    FileTransferMethodId: 0
    TemplateParameterValues:
      computationPrecision: { get_input: precision }
    EnvironmentVariables: {}
  requirements:
  - heappe_session_task_id: 1_heappe_session
1_heappe_submit_job:
  type: lexis.nodes.operators.heappe.HEAppESubmitJobOperator
  properties:
    heappe_session_task_id: {
        get_attribute: [ 1_heappe_session,task_id ]
    }
    heappe_prepare_job_task_id: {
        get_attribute: [ 1_heappe_prepare_job,task_id ]
    }
  requirements:
  - heappe_session_task_id: 1_heappe_session
  - heappe_prepare_job_task_id: 1_heappe_prepare_job
  - previous_task: 1_heappe_prepare_job
1_heappe_job_wait:
  type: lexis.nodes.operators.heappe.HEAppEWaitSubmittedJobSensor
  properties:
    heappe_session_task_id: {
        get_attribute: [ 1_heappe_session,task_id ]
    }
    heappe_submitted_job_task_id: {
        get_attribute: [ 1_heappe_submit_job,task_id ]
    }
  requirements:
  - heappe_session_task_id: 1_heappe_session
```

```
  - heappe_submitted_job_task_id: 1_heappe_submit_job
1_heappe_delete_job:
  type: lexis.nodes.operators.heappe.HEAppEDeleteJobOperator
  properties:
    heappe_session_task_id: {
        get_attribute: [ 1_heappe_session,task_id ]
    }
    heappe_prepare_job_task_id: {
        get_attribute: [ 1_heappe_prepare_job,task_id ]
    }
  requirements:
  - heappe_session_task_id: 1_heappe_session
  - heappe_prepare_job_task_id: 1_heappe_prepare_job
  - previous_task: 1_heappe_job_wait
1_heappe_session_keeper:
  type: lexis.nodes.operators.heappe.HEAppESessionKeeperSensor
  properties:
    heappe_session_task_id: {
        get_attribute: [ 1_heappe_session,task_id ]
    }
    heappe_session_stop_task_id: {
        get_attribute: [ 1_heappe_session_stop_keeper,task_id ]
    }
  requirements:
  - heappe_session_task_id: 1_heappe_session
1_heappe_session_stop_keeper:
  type: lexis.nodes.operators.heappe.HEAppESessionStopKeeperOperator
  requirements:
  - previous_task: 1_heappe_delete_job
aai_token_keeper:
  type: lexis.nodes.operators.aai.LexisAAITokenKeeperSensor
  requirements:
  - aai_operator: aai_token_init
aai_token_keeper_stop:
  type: lexis.nodes.operators.aai.LexisAAIStopKeeperOperator
  requirements:
  - previous_task: 1_heappe_session_stop_keeper
```

Appendix C

User Guide

C.1 Build instruction

To run the application, you will need to install the .NET 7 framework Blazor server. You can follow the .NET Blazor Tutorial `https://dotnet.microsoft.com/en-us/learn/aspnet/blazor-tutorial/intro` for instructions on how to install the framework.

The build process for the application depends on the host operating system. For development, I used Linux (OpenSUSE) and installed the `dotnet` package for .NET 7 from my package provider. Once the framework was installed, I opened the terminal and navigated to the `Sources/builderApp` directory. From there, I ran the command `dotnet build` and the binary executable was created at
`Sources/builderApp/bin/Debug/net7.0/scientific-workflow-gui`.

C.2 Usage instruction

The .NET project options can be modified in the file
`Sources/builderApp/scientific-workflow-gui.csproj`. If the user builds the application without modifying any options and executes the compiled binary, the web application will be exposed on the URL `http://localhost:5000/WorkflowEditor`. The application runs locally, and users can compose simple workflows from predefined computational tasks.

To execute and host the application locally, navigate to the `Sources/builderApp` directory in the terminal and run the command `dotnet run`. In this case, the application will be listening on `http://localhost:5039/WorkflowEditor`.

The user can add predefined computational tasks to the workflow diagram by clicking on the plus button located in the right navigation menu. To add dataset inputs, the user can click on the button located at the bottom of the workflow input menu, which can be accessed by clicking on the second button in the right navigation menu. To link a task's data input and output, the user can click on the left mouse button on the port of the computational task and release it on the port where the connection is intended. In the workflow input menu, the user can rename data inputs and set default values. To export the computational task's data output, the user can open the task's editing menu by right-clicking on the task in the diagram and selecting *Edit Task*. From there, the user can export the data output.

Appendix D

Application Demo 1

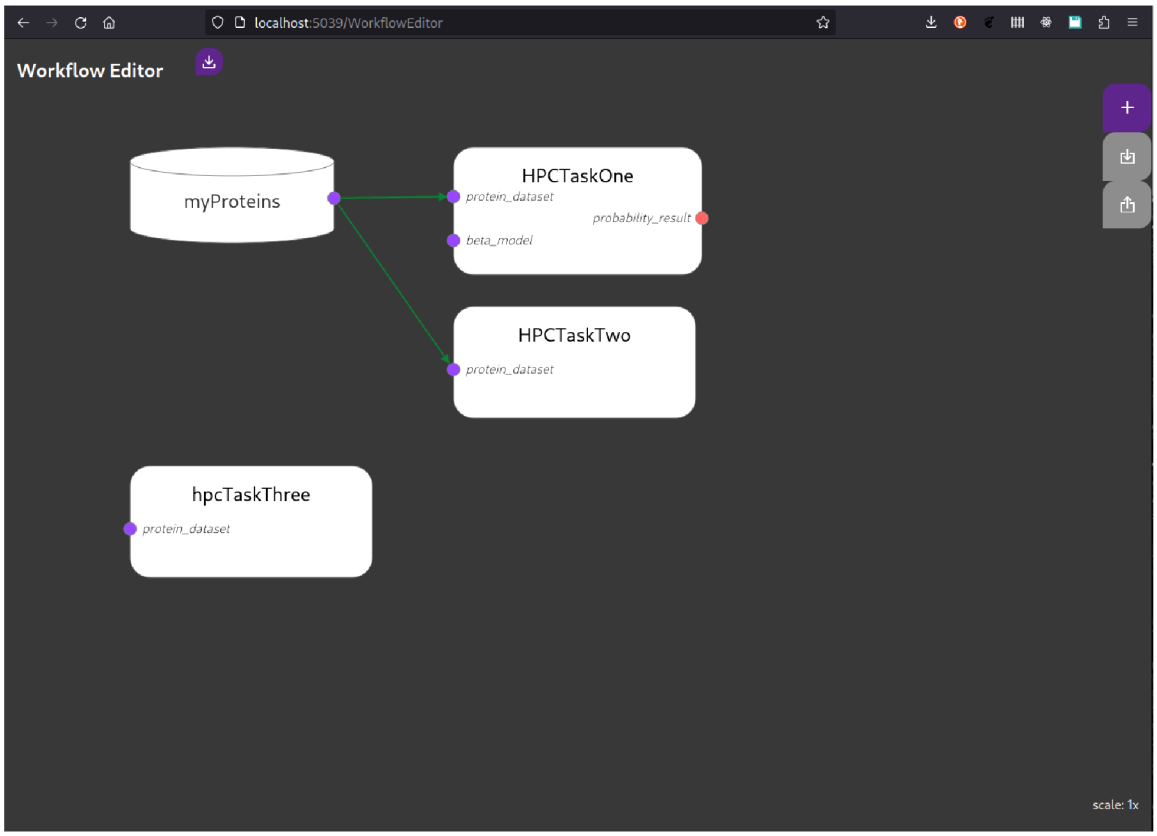


Figure D.1: Application Demo 1 – Workflow editor interface

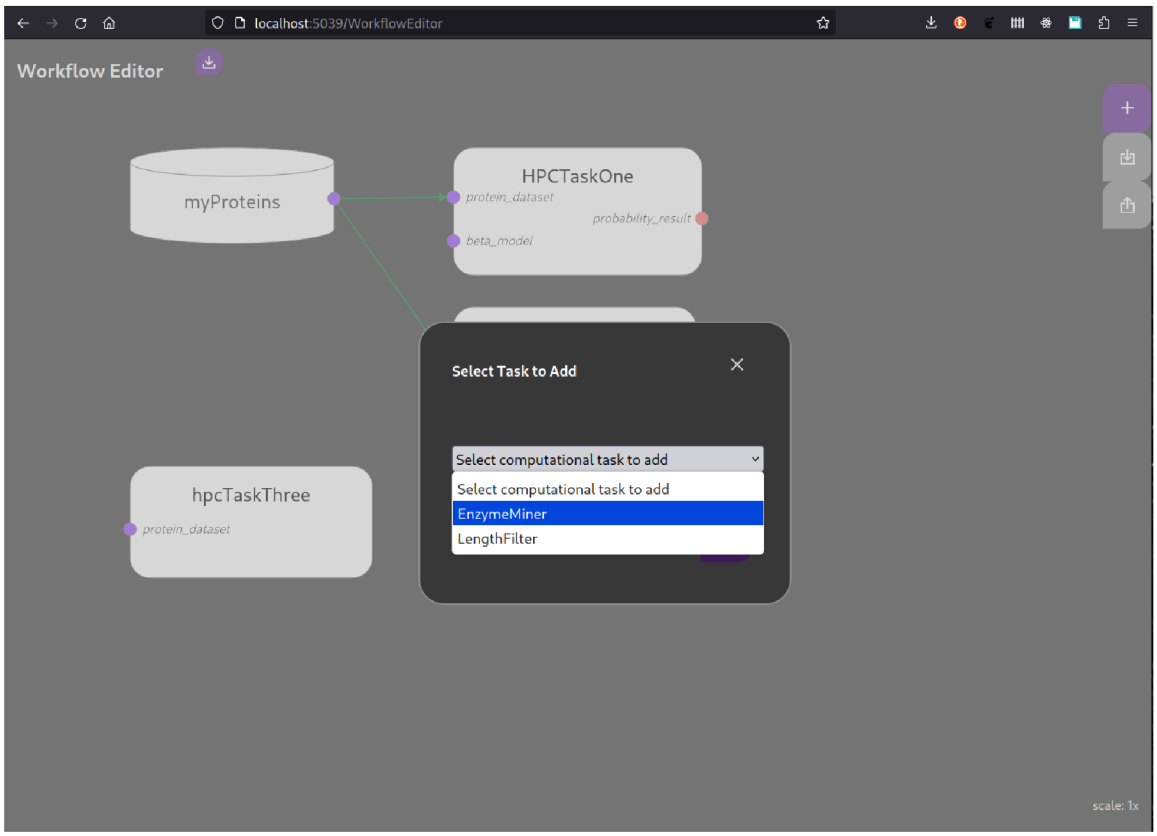


Figure D.2: Application Demo 1 – Modal window for adding a computational task to the diagram

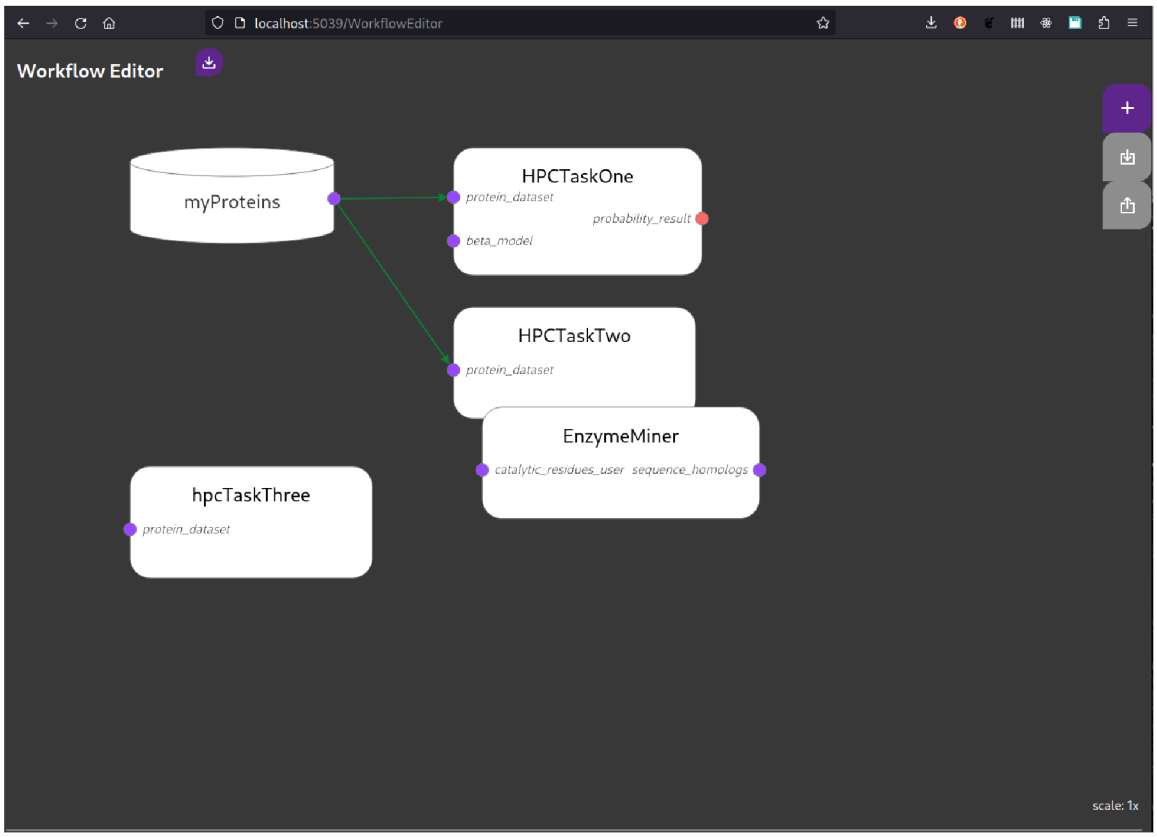


Figure D.3: Application Demo 1 – New computational task added to diagram

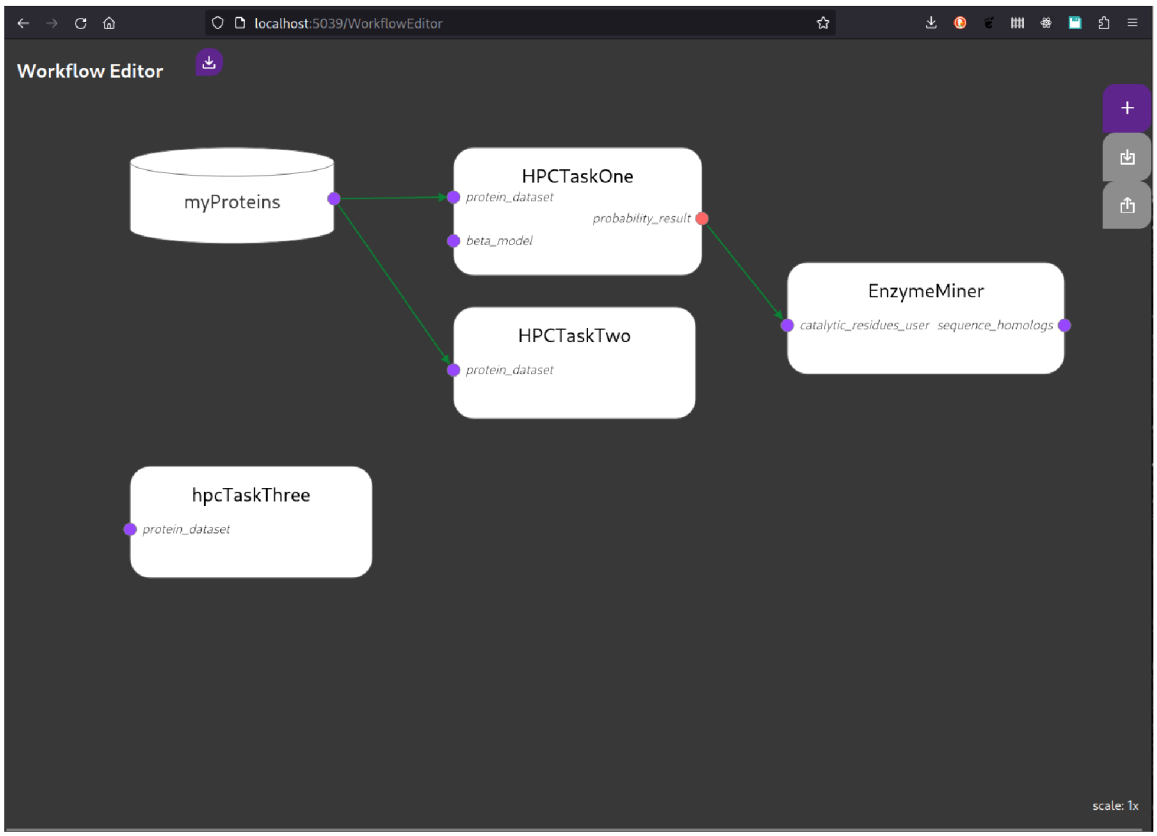


Figure D.4: Application Demo 1 – *HPCTaskOne* task's data output connected to *EnzymeMiner* task's data input in the diagram

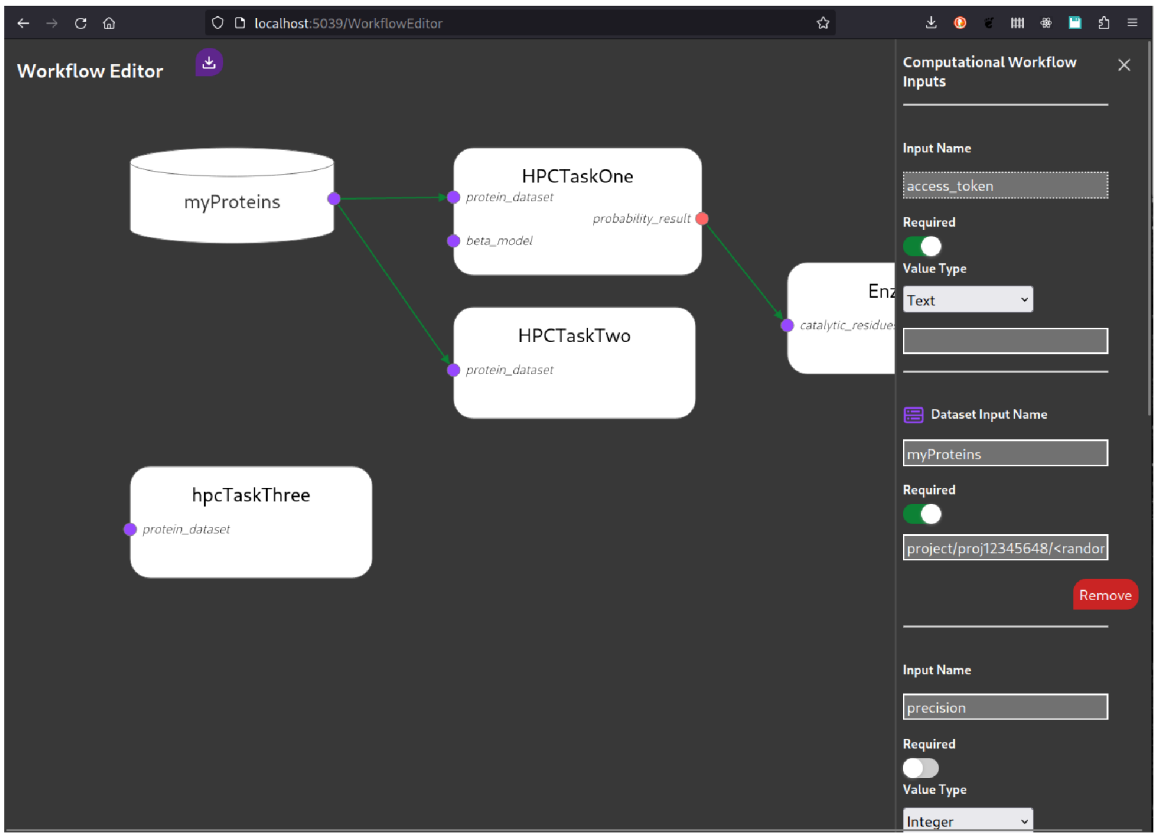


Figure D.5: Application Demo 1 – Screenshot of the opened workflow's input menu

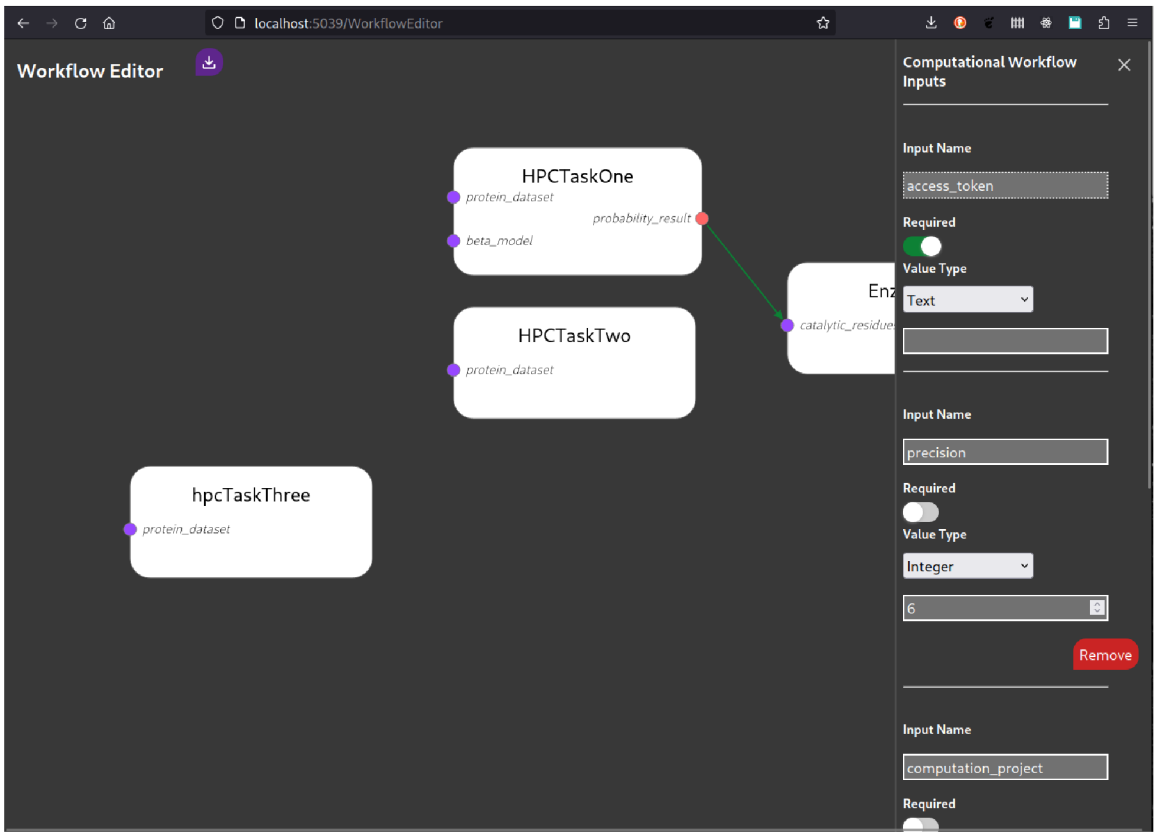


Figure D.6: Application Demo 1 – Editor's screenshot with removed *myProteins* workflow data input from workflow's input menu

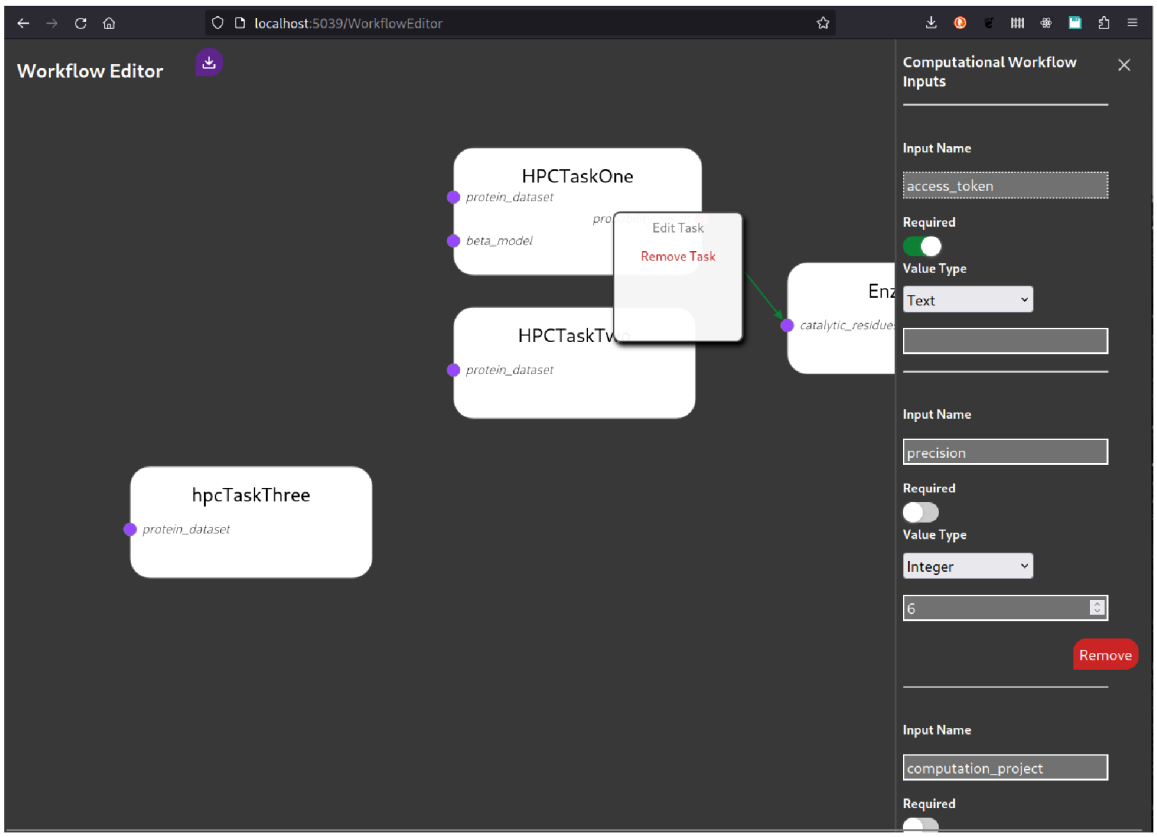


Figure D.7: Application Demo 1 – Computational task's context menu

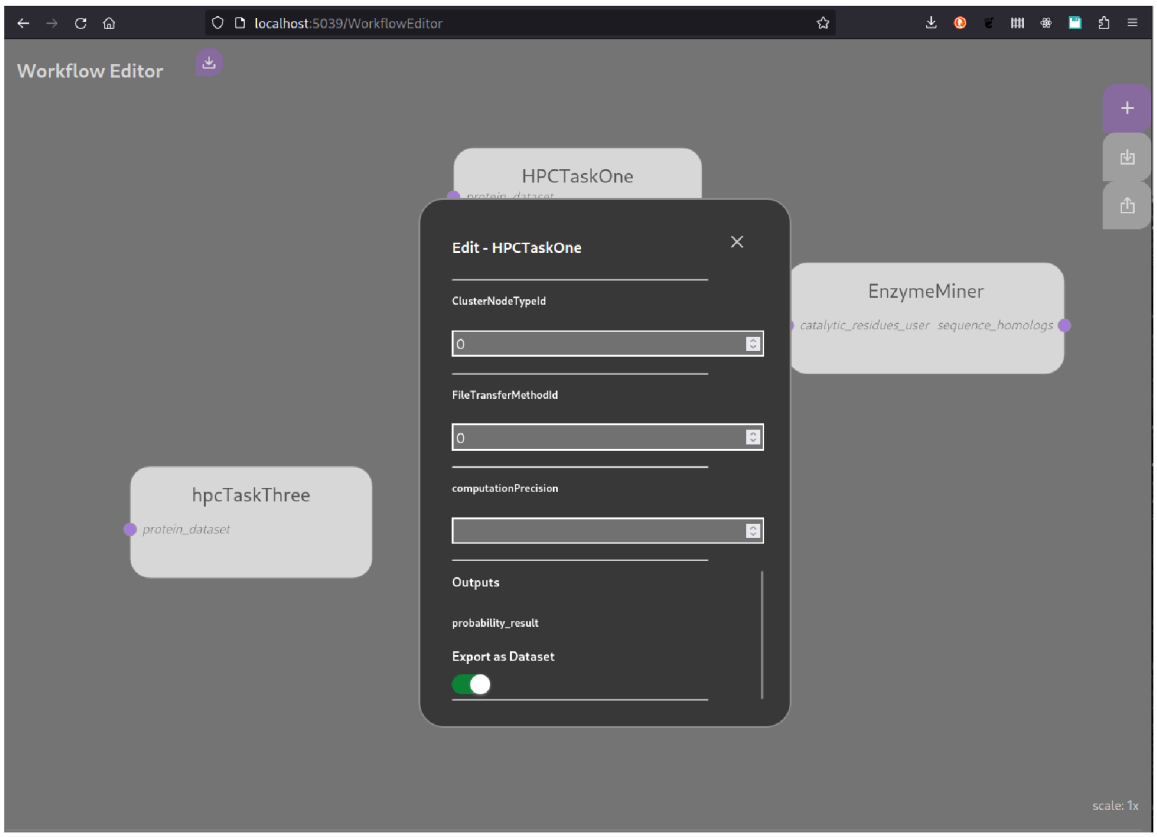


Figure D.8: Application Demo 1 – Modal window for editing computational task

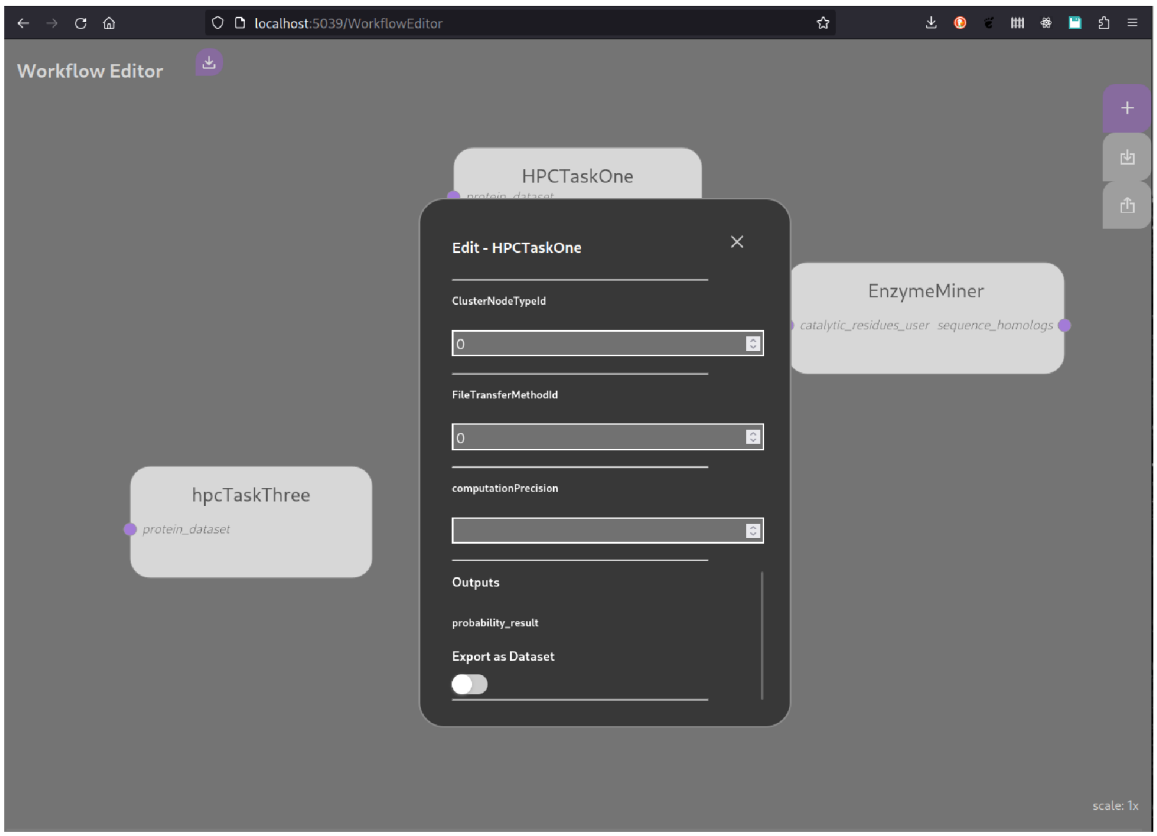


Figure D.9: Application Demo 1 – Modal window for editing computational task with disabled export of data output

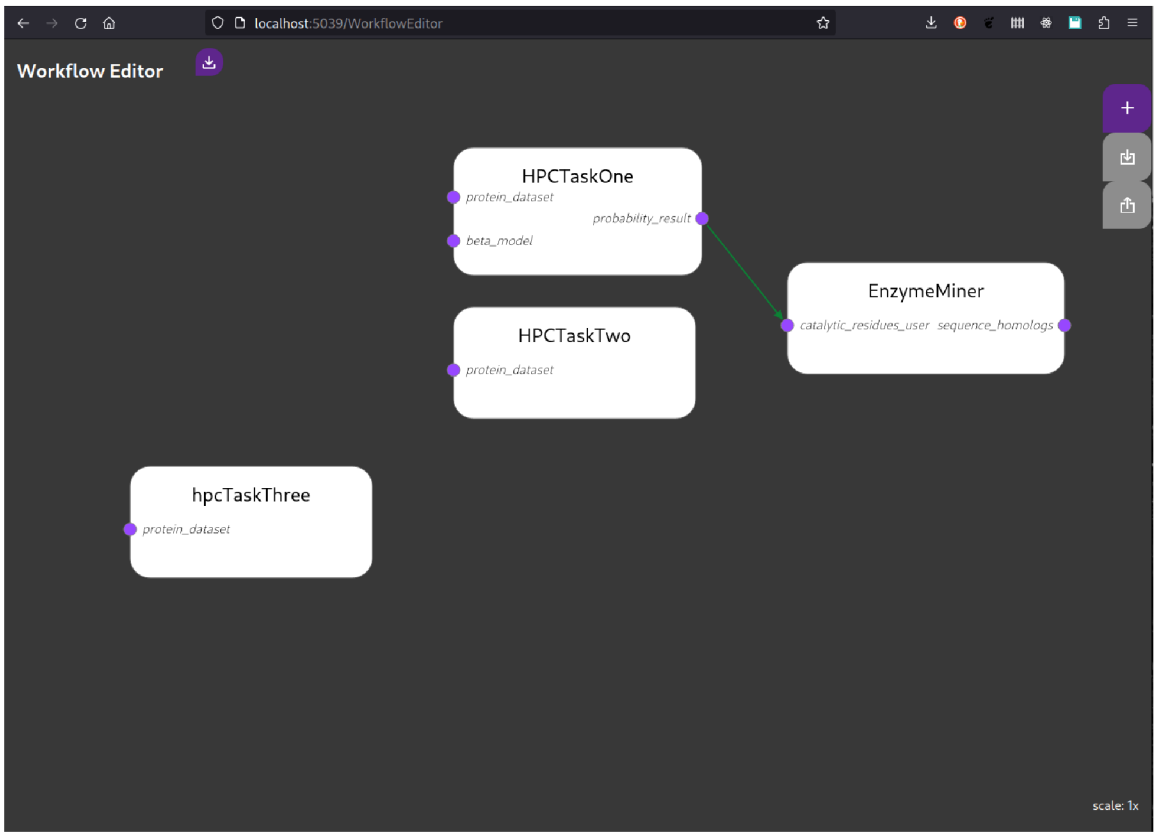


Figure D.10: Application Demo 1 – *HPCTaskOne* computational task has disabled export of data output

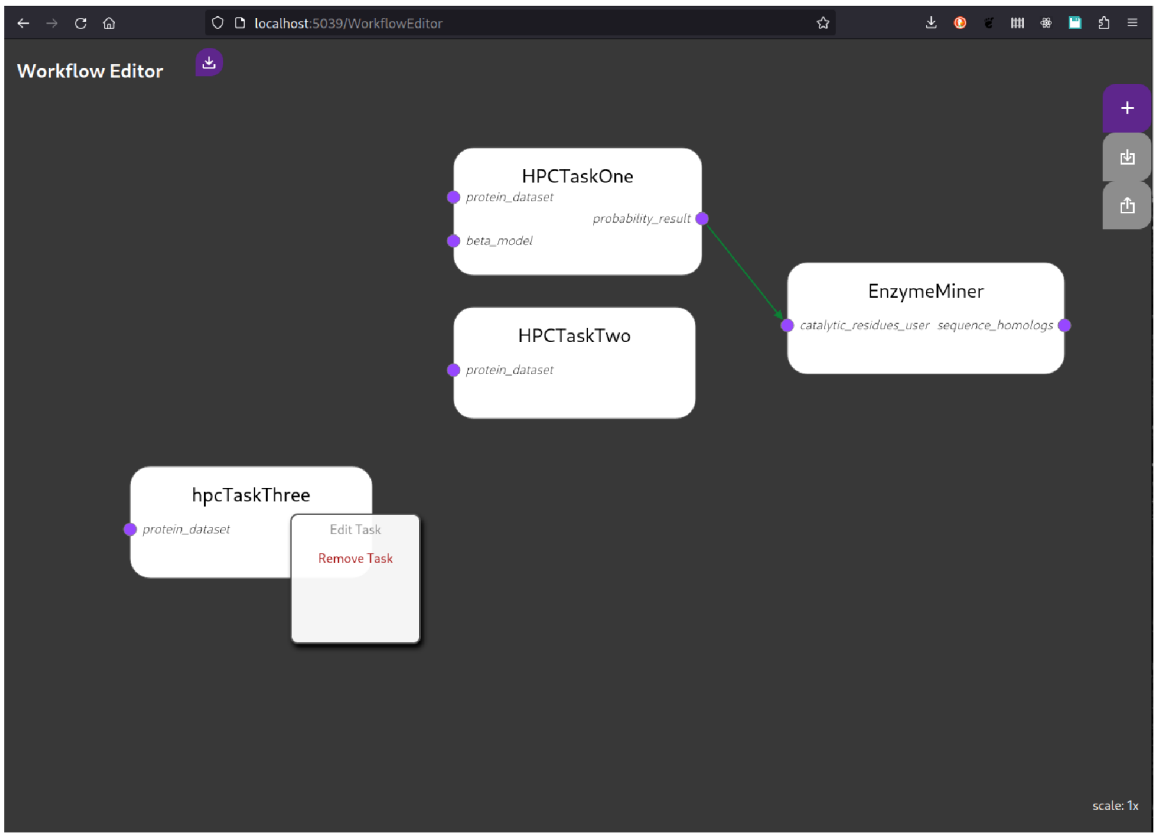


Figure D.11: Application Demo 1 – Removing computational task *hpcTaskThree*

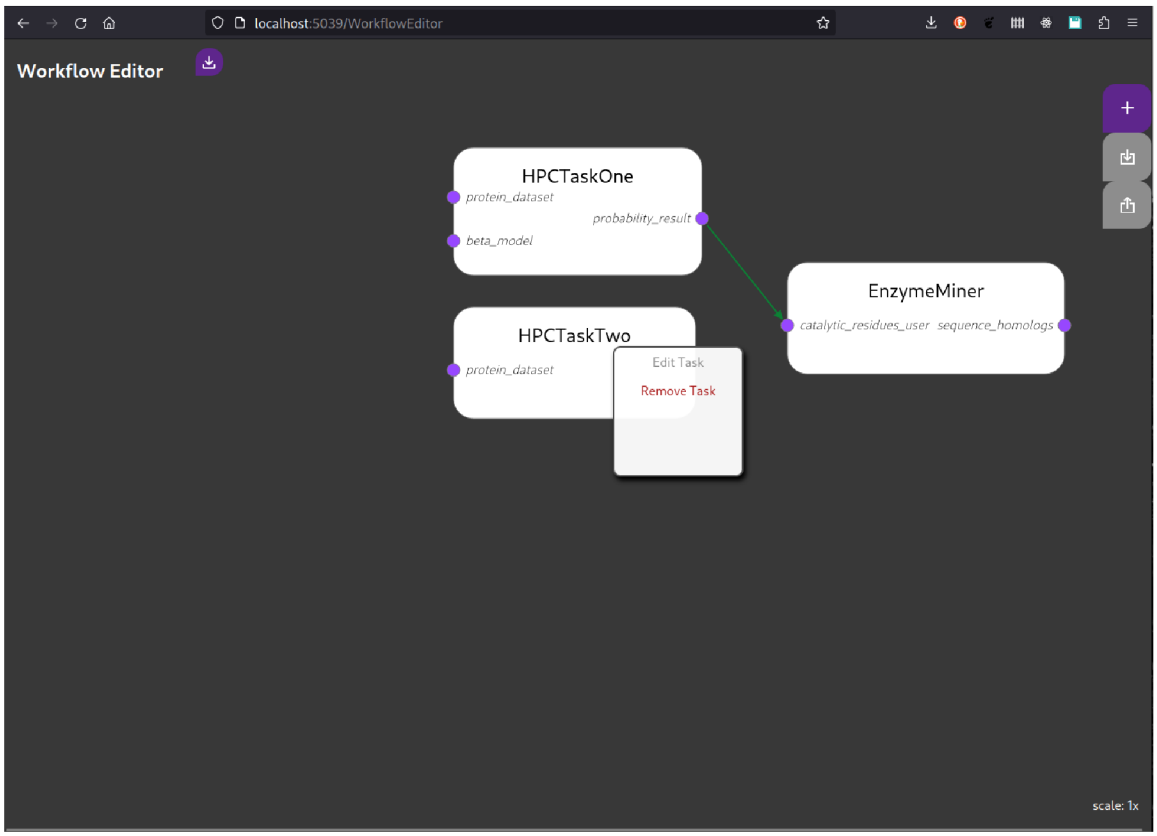


Figure D.12: Application Demo 1 – The TOSCA workflow specification is downloaded via browser after clicking on the download button next to the title

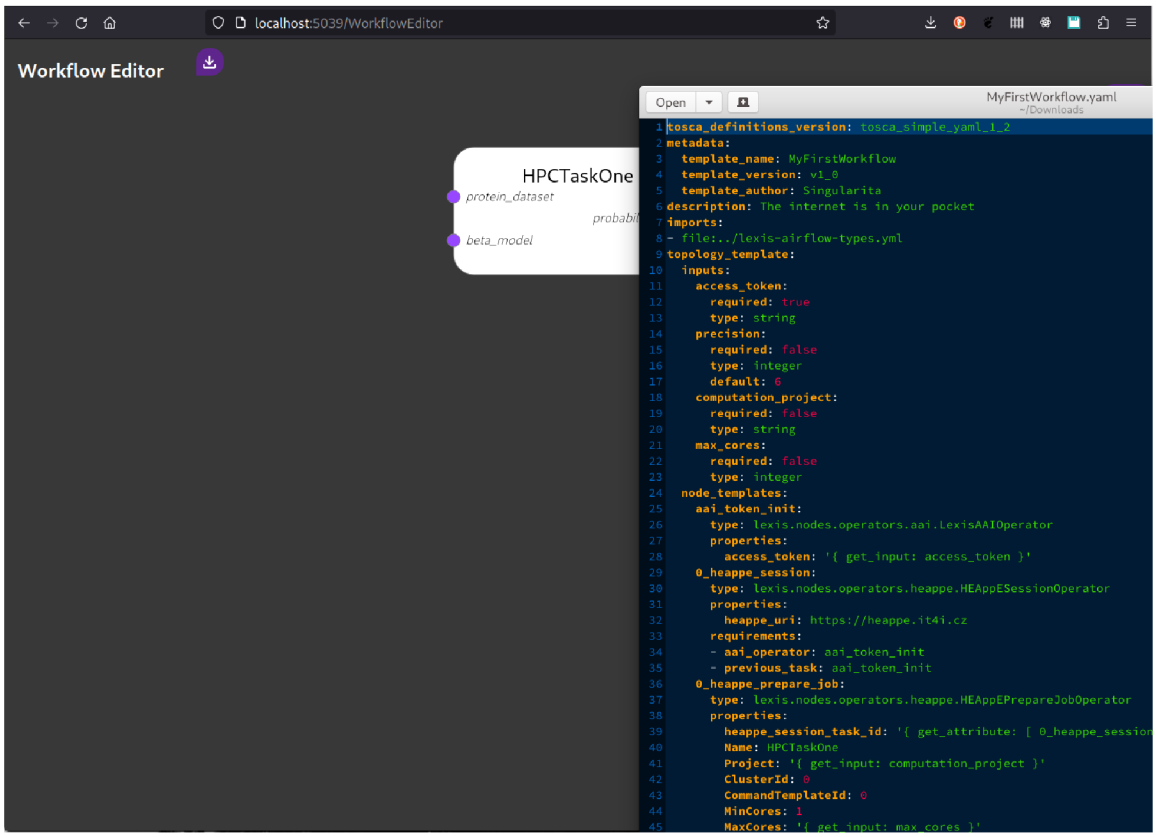


Figure D.13: Application Demo 1 – Exported TOSCA specification

Appendix E

Application Demo 2

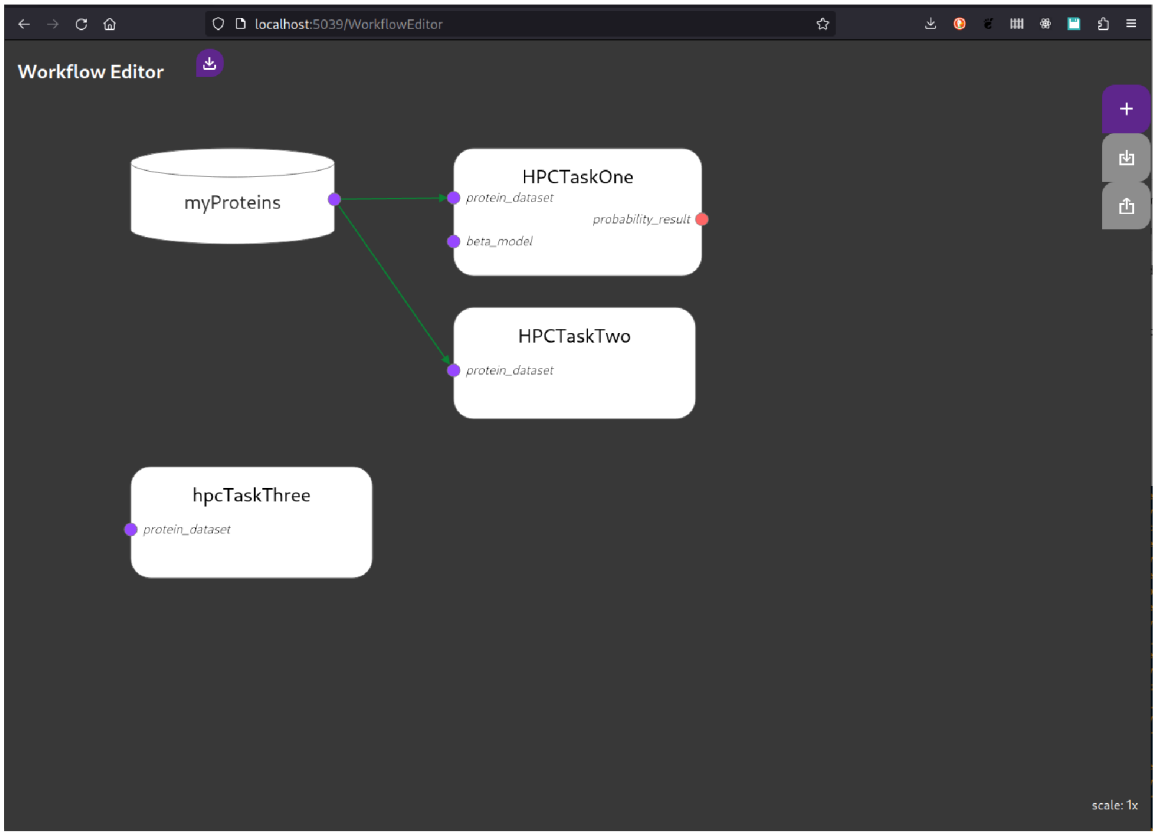


Figure E.1: Application Demo 2 – Workflow editor interface

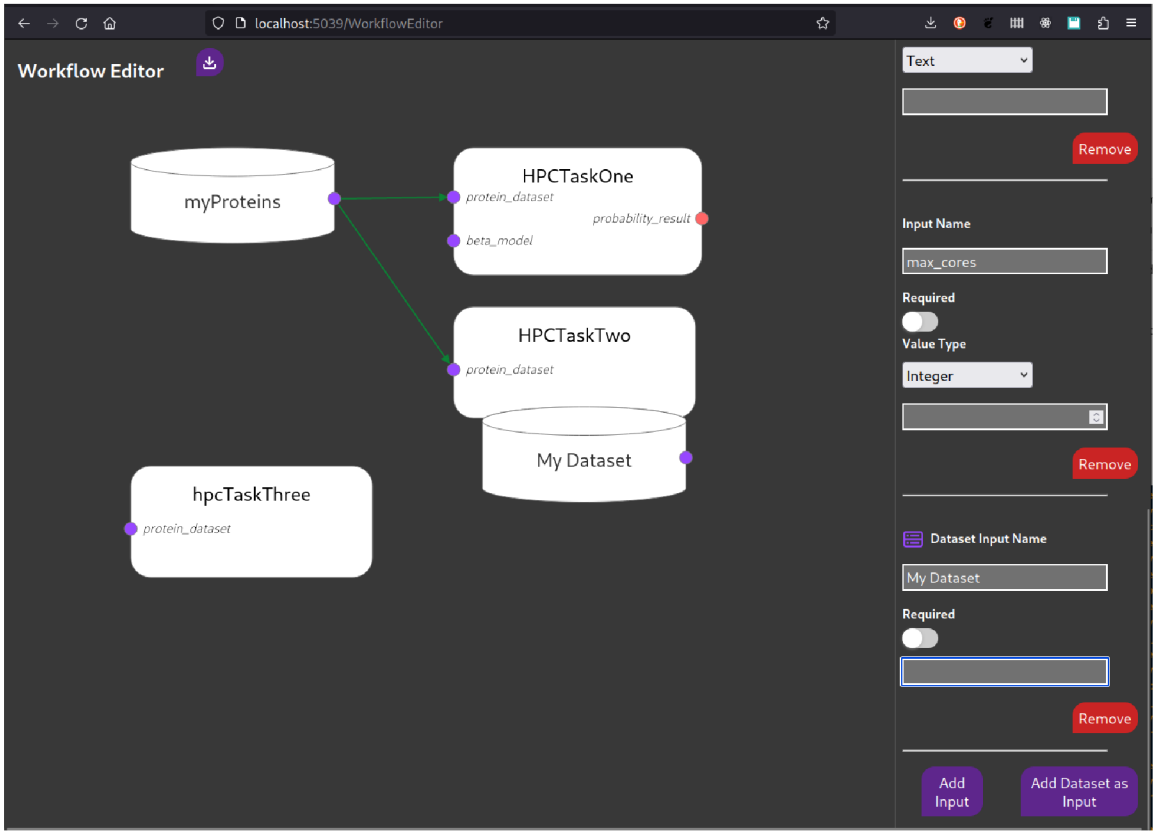


Figure E.2: Application Demo 2 – New workflow's dataset input added to the workflow

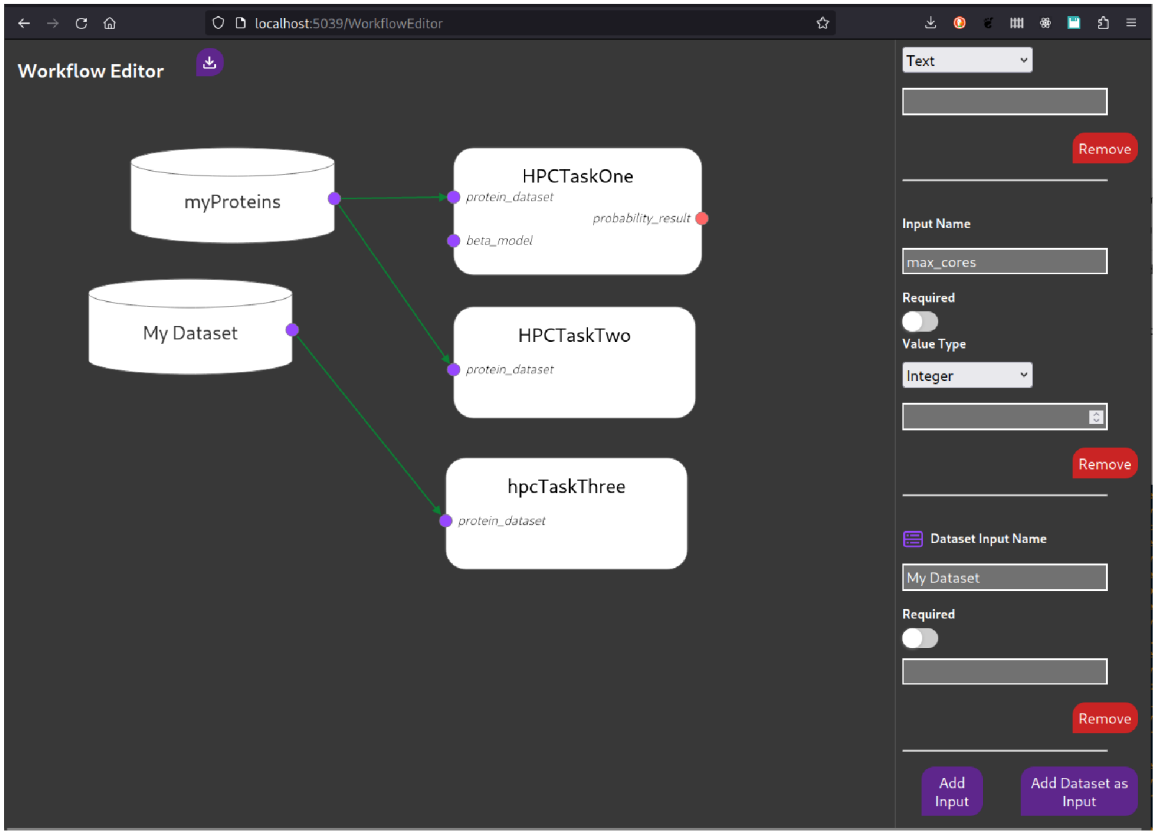


Figure E.3: Application Demo 2 – New workflow's dataset input connected to computational task's input