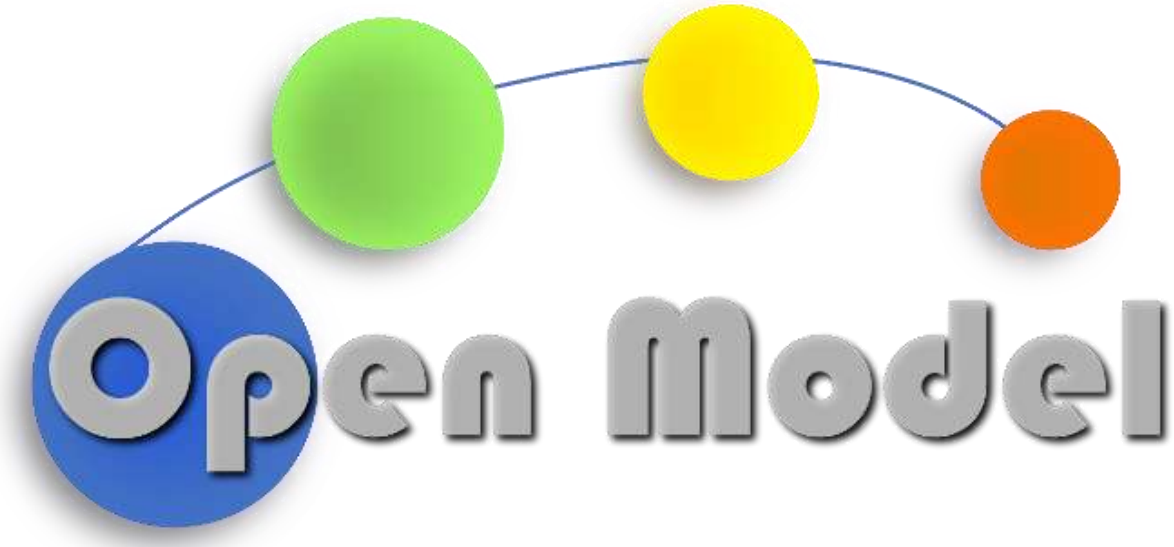
2023



LOUIS PONET

| Rte Cantonale, 1015 Lausanne

D 4.5: DEMONSTRATION OF EXECFLOW FOR AT LEAST ONE FULL WORKFLOW

DOCUMENT CONTROL

Document Type	Deliverable Report
Status	Initial draft
Version	0.1
Responsible	Louis Ponet (EPFL)
Author(s)	Louis Ponet (EPFL), Casper W. Andersen (SINTEF), Thomas Hagelien (SINTEF)
Release Date	31-01-2023

ABSTRACT

OpenModel utilizes AiiDA as its workflow executor. To facilitate the dynamic and changeable nature of workflows that will be designed and run based on the needs of the end user, a flexible method of running WorkChains is developed based on two “meta” WorkChains. The first is the **`DeclarativeChain`**, which orchestrates the overall execution of the workflow, while the second is the **`DeclarativePipeline`**, which allows to execute an OTEAPI pipeline in AiiDA, keeping track of the full data provenance. These can be described in structured text-based files (**`yaml`**, **`json`**, etc), with a simple syntax similar to **`GitHub Actions`** or **GitLab Pipelines**. The end result is a method that allows for developing and running novel AiiDA WorkChains with much less friction than the standard method with potential impact far beyond the scope of OpenModel. In the specific case of OpenModel, it is OntoFlow that will generate the text representation after having designed the workflow based on the end user’s needs.

CHANGE HISTORY

Version	Date	Comment
0.1	2023-01-19	First Draft

0.2	2023-01-27	First review by Technical Coordinator
0.3	2023-01-30	Added description of declarative pipelines
0.4	2023-01-31	Final review by technical coordinator
1.0	2023-01-31	Final

DISSEMINATION LEVEL

PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

TABLE OF CONTENT

1 INTRODUCTION

The two main parts of OpenModel that involve workflows, which are comprised of a series of simulations and data pre- and post-processing steps, are OntoFlow and ExecFlow. OntoFlow handles the creation of the workflow by reasoning about the necessary steps to achieve the final result. This will depend on the user's requirements, available simulations, and previously gathered data in the knowledge base. The task of ExecFlow is then to take the generated workflow and execute it with AiiDA as the execution engine. This generally also involves fetching and storing data from the CUDS database that is used in OpenModel. The nature and number of workflows that will be run in the scope of OpenModel requires a way to communicate the workflows which is more flexible and dynamic than AiiDA currently allows for. WorkChains are currently represented as Python Classes, as AiiDA is built in Python. The reason for using Classes is that this ideally leads to WorkChains which are well-defined, robust, and reproducible. One of the consequences of this design choice, however, is that the AiiDA daemon can only run WorkChains that are known and registered at the time of daemon startup. This means that any new workflow designed and submitted from OntoFlow to ExecFlow would require stopping the AiiDA daemon, generating the WorkChain Python Class, registering it, restarting the daemon, and finally submitting the WorkChain. This is an inefficient, clunky, and most importantly brittle process: any change to how WorkChains have to be implemented would break all previously defined WorkChains, making them by definition only reliably reproducible for one single version of AiiDA. While this process works for more generalized WorkChains, it is far from ideal in the more dynamic context OpenModel strives to achieve.

Here we instead decided to go a different route, and develop a single "meta"- WorkChain, named **`DeclarativeChain`**, which can ingest a workflow specification to self-assemble the steps that comprise the workflow on the fly. This has allowed us to develop a very minimal syntax-based representation of the workflow in a structured format using **`yaml`** or **`json`** files. In fact, the result is a novel method that allows for developing and running a novel AiiDA WorkChain with less friction than the standard method with potential impact far beyond the scope of OpenModel. Since the interface to AiiDA is now effectively a structured text file, any software that can output such a file can use AiiDA as its execution engine. In the case of OpenModel this software is OntoFlow.

While data that is to be used in the workflow can be fully specified in the workflow specification, in the framework of OpenModel it is desirable to ingest CUDS stored in a triple store or knowledge base. To facilitate this, we developed a way to execute OTEAPI pipelines as AiiDA WorkChains, using similar concepts to those previously described, leading to the **`DeclarativePipeline`**.

The implementation of the work performed in this deliverable can be found in the OpenModel GitHub repository: https://github.com/H2020-OpenModel/WP4-documents/. Some of the work has also been contributed directly to AiiDA.

2 DECLARATIVE WORKCHAIN

As mentioned above, the **DeclarativeChain** is a self-assembling AiiDA WorkChain. It takes as its input a **SingleFileData** which represents an on-disk text-based representation of the workflow. Currently, this can be either a **`json`** or **`yaml`** file. A detailed discussion of the syntax in these files will be presented below. In essence, every WorkChain consists of a sequence of steps, each of them representing a block of computationally non-trivial work which transforms input data to output data. The **DeclarativeChain** reads these steps from the input file and internally translates them into AiiDA Processes and runs them, using some directives to allow for communicating dataflow between each of the steps, thereby fully tracking the provenance as usual in AiiDA.

2.1 SYNTAX

We will discuss the syntax of the input file to the **DeclarativeChain** using the **`yaml`** format, but nothing changes when using **`json`** except the look of the file, since the JSON syntax is a subset of the YAML syntax. The most basic structure of the file is:

```
---
steps:
    - <step-type>: <step-type-specifier>
      <step-configuration>
```

Where **`step-type`** and **`step-type-specifier`** can be:

- **`calcjob`**: entrypoint to AiiDA CalcJob or CalcFunction, e.g. **`calcjob: quantumespresso.pw`**
- **`workflow`**: entrypoint to AiiDA WorkChain or WorkFunction, e.g. **`workflow: quantumespresso.pw.bands`**
- **`while`**: jinja template for while loop statement, e.g. **`while: {{ ctx.count < 3 }}`**
- **`if`**: jinja template for if statement, e.g. **`if: {{ ctx.count < 3 }}`**

The former two represents an AiiDA calculation and workflow step, respectively. The **`while`** step will execute the body of the step while the statement is true, whereas the **`if`** step will only execute when the if statement is true. These latter two demonstrate the use of **`jinja`** templates, which allow for the execution of some code in a sandboxed manner. Essentially, they will dynamically replace the code in braces with the result and insert it before parsing the **`step`**. In the examples, they look at the **`count`** variable stored in the context (**`ctx`**) object of the workchain and insert **`true`** if it is lower than 3 and **`false`** if not.

The **`calcjob`** or **`workflow`** steps consist of 4 parts behind the scenes:

- Initialization of an AiiDA Process representing the computational task.
- Retrieval of the inputs for the Process, either from inside the context of the **DeclarativeChain** (e.g. data that was previously saved to this context, more on this later), or from a textual representation inside the **DeclarativeChain** input file.
- Submitting or running the Process. The first being asynchronous and non-blocking, while the latter is a synchronous process and blocking for the given computational thread.
- Postprocessing, in which one can extract outputs of a step to be stored into the context of the **DeclarativeChain** to be used later or attach them to the outputs/results of the **DeclarativeChain**.

The configuration has the following structure:

```
- calcjob: quantumespresso.pw
  inputs:
    <input name 1>: <data>
    <input name 2>: <data>
  postprocess:
    - <postprocess 1>
    - <postprocess 2>
```

The **inputs** dictionary is a one-to-one representation of the input Nodes to the AiiDA CalcJob or WorkChain. Behind the scenes, the *<data>* will be converted into the right AiiDA Data Node, provided that the CalcJob or WorkFlow defines what it should be. If this is not the case, using the *type* key in the following syntax can be used to manually declare what Data Node type should be created:

```
string_input:
    value: "demo"
    type: orm.str
```

This means that *"demo"* will be turned into an AiiDA Str Node before being attached as the *"string_input"* input to the CalcJob.

There are two more ways data can be referenced as an input. The first is by using **jinja** templates to extract it from the **ctx** of the WorkChain (this requires it to be defined in the first place, more on this later):

```
inputs:
  string_input: {{ctx.demo}}
```

The second is using **jsonref** to reference data that is defined somewhere else in the file, or even in another file (see **jsonref** documentation for more information). In the simplest case this looks like:

```
data:
    string_input: "demo"
steps:
- calcjob: quantumespresso.pw
  inputs:
    string_input:
        "$ref": "#/data/string_input"
```

The postprocess step consists of a list of **jinja** templates that can be executed, potentially using the two custom **jinja** filters: **to_ctx** and **to_results:**

```
postprocess:
  - {{ ctx.current.outputs['output_link1']|to_ctx('output1') }}
  - {{ ctx.current.outputs['output_link2']|to_results('result1') }}
```

In the first postprocessing step, we take the output link *'output_link1'* from the CalcJob of the step (represented by **ctx.current**), and store it in the *'output1'* variable. This means that later we can reference it as an input to another step. The second step means that we take *'output_link2'* and attach it to the outputs of the **Declarative-Chain** as a whole. These two directives supply all the glue code that is required to string together steps in a workflow.

The **while** and **if** step types have a body with a single list of steps, with the same structure as the **Declarative-Chain** as a whole:

```
steps:
- <if/while>: {{ ctx.count < 3}}
    steps:
        - calcjob: <entrypoint>
          inputs:
            string_input:
                "$ref": "#/data/string_input"
          postprocess:
            - {{ ctx.current.outputs['output_link1']|to_ctx('output1') }}
            - {{ ctx.current.outputs['output_link2']|to_results('result1') }}
```

Now, there is an issue with the above example. We are referencing a context variable without having defined it, since so far we only discussed the definition of context variables as part of the postprocessing step. For such cases, there is one other special top-level entry: **`setup`**. This has essentially the same behavior as **`postprocess`** but it is executed when the WorkChain starts. In this case it would look like the following:

```
---
data:
    string_input: "demo"
setup:
    - {{ 2 | to_ctx('count')}}
steps:
- <if/while>: {{ ctx.count < 3}}
    steps:
        - calcjob: <entrypoint>
          inputs:
            string_input:
                "$ref": "#/data/string_input"
          postprocess:
            - {{ ctx.current.outputs['output_link1']|to_ctx('output1') }}
            - {{ ctx.current.outputs['output_link2']|to_results('result1') }}
```

This describes all the necessary syntax to run **DeclarativeChains**.

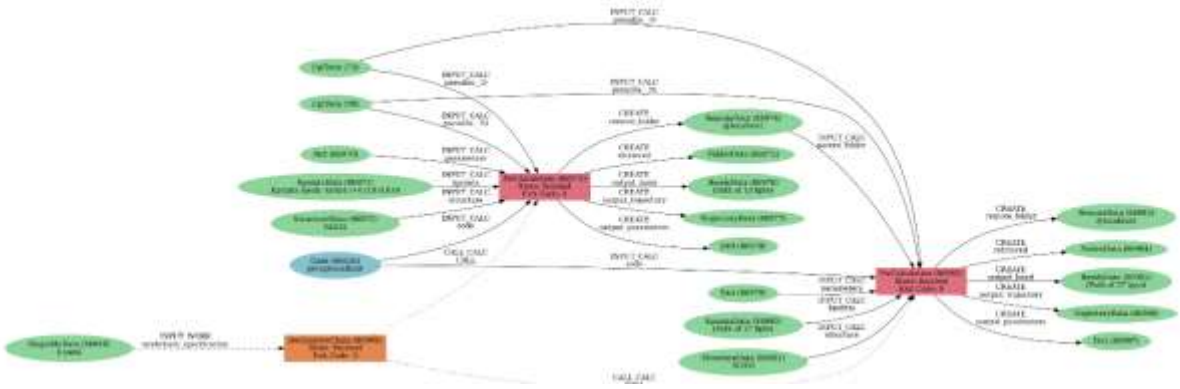
2.2 DEMONSTRATING EXAMPLE

Working examples can be found in WP4-ExecFlow Demos[1]. The resulting provenance graph of connecting a QuantumEspresso[2] scf with a nscf[3] calculation is as follows:

[1] https://github.com/H2020-OpenModel/WP4-documents/tree/master/ExecFlow/Demo

[2] Quantum Espresso is a density functional theory software package (https://www.quantum-espresso.org). It has a preexisting AiiDA plugin: https://github.com/aiidateam/aiida-quantumespresso

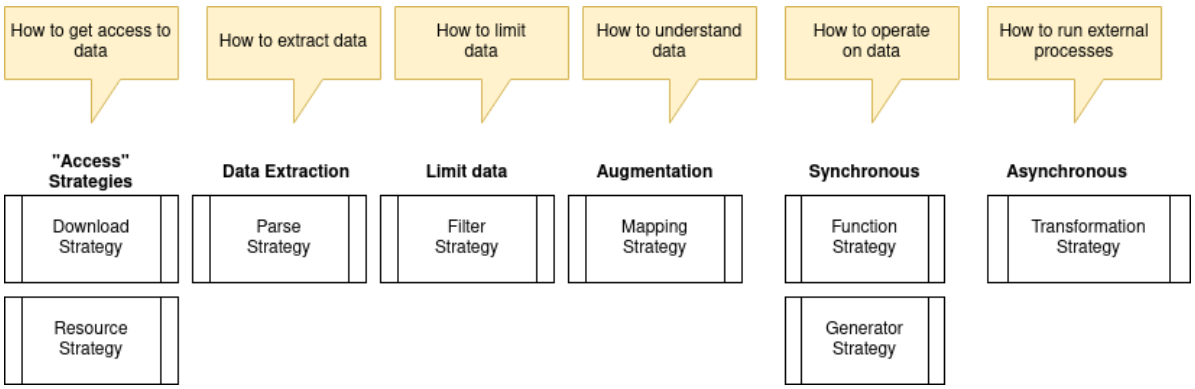
[3] An scf calculation finds the correct electronic wavefunctions and charge density associated with the ground state of a physical system



A more complicated example demonstrating the full syntax capabilities and its output can be found as *6.yaml* and *6.pdf* in the same directory.

3 DECLARATIVE PIPELINE

A data pipeline allows for defining the transformation of data from a data source to a consumer. The pipeline is defined by a set of data models that document how data can be accessed, extracted, filtered, transformed, and documented in terms of a common vocabulary. In the ExecFlowSDK (software development kit) the pipeline is built on the OTEAPI-pipelines[4] and adopted the AiiDA execution environment. In the execution of a pipeline, a set of strategies will operate on the configuration and perform individual steps. For instance, a download strategy will read a *downloadUrl* from the configuration and fetch the artifact. The configuration of a pipeline can, similarly to the **DeclarativeChain**, be defined as a single **JSON** or **YAML** file. In the **DeclarativePipeline**, each element of the pipeline is configured, and a set of pipelines can be defined as a list of the defined elements. The OpenModel ExecFlowSDK will parse the **DeclarativePipeline** to configure an AiiDA WorkChain that can be executed in the same environment as and by the **DeclarativeChain**. AiiDA will upon execution of the pipeline generate the provenance graph for each step in the pipeline.



[4] Documentation of the OTEAPI https://EMMC-ASBL.github.io/oteapi-core/

Each elementin the pipeline belongs to a specific category. In the figure above, the different categories (elementtypes) are illustrated with an explanation of their purpose. OTEAPI is based on plug-ins, where a set of strategies implement the functionality of a specific element. Each strategy defines a generic data model for the elementtype, along with a specific configuration. In the AiiDA wrappers for the pipeline elements, these configurations are implemented as AiiDA Data Nodes. In the **YAML/JSON** declaration of the pipeline, these data models will dictate the accepted configurations.

The basic structure of the **DeclarativePipeline** is defined as follows:

```yaml
version: "1"
strategy:
  - <element>: <element-name>
      .. configuration

  - <element>: <element-name>
      .. configuration

pipeline:
  <pipeline-name>: <element-name> | <element-name>

```

The **YAML** file consist of a series of configurations for each strategy. This corresponds to the OTEAPI documentation. A pipeline is constructed by defining a named list of pipeline elements.

3.1 EXAMPLE OF RUNNING A DECLARATIVE PIPELINE IN AIIDA

Consider the following pipeline example from the ExecFlowSDK documentation. Here we define two pipeline elements; *dataresource* and *mapping*. The data resource is defined with a *downloadUrl* and a *mediaType*. This configuration will allow OTEAPI to decide which strategy (plugin) that will be executed. This file is stored in the local filesystem. Here we call it ***pipe.yml***.

```yaml
version: 1

strategies:
  - dataresource: json_file
    downloadUrl: "https://filesamples.com/samples/code/json/sample2.json"
    mediaType: application/json

  - mapping: map_json_file
    mappingType: triples
    prefixes:
      map: "http://example.org/0.0.1/mapping_ontology#"
      onto: "http://example.org/0.2.1/ontology#"
    triples:
      - ["http://onto-ns.com/meta/1.0/Foo#a", "map:mapsTo", "onto:A"]
      - ["http://onto-ns.com/meta/1.0/Foo#b", "map:mapsTo", "onto:B"]
      - ["http://onto-ns.com/meta/1.0/Bar#a", "map:mapsTo", "onto:C"]

pipelines:
  pipe: json_file | map_json_file
```

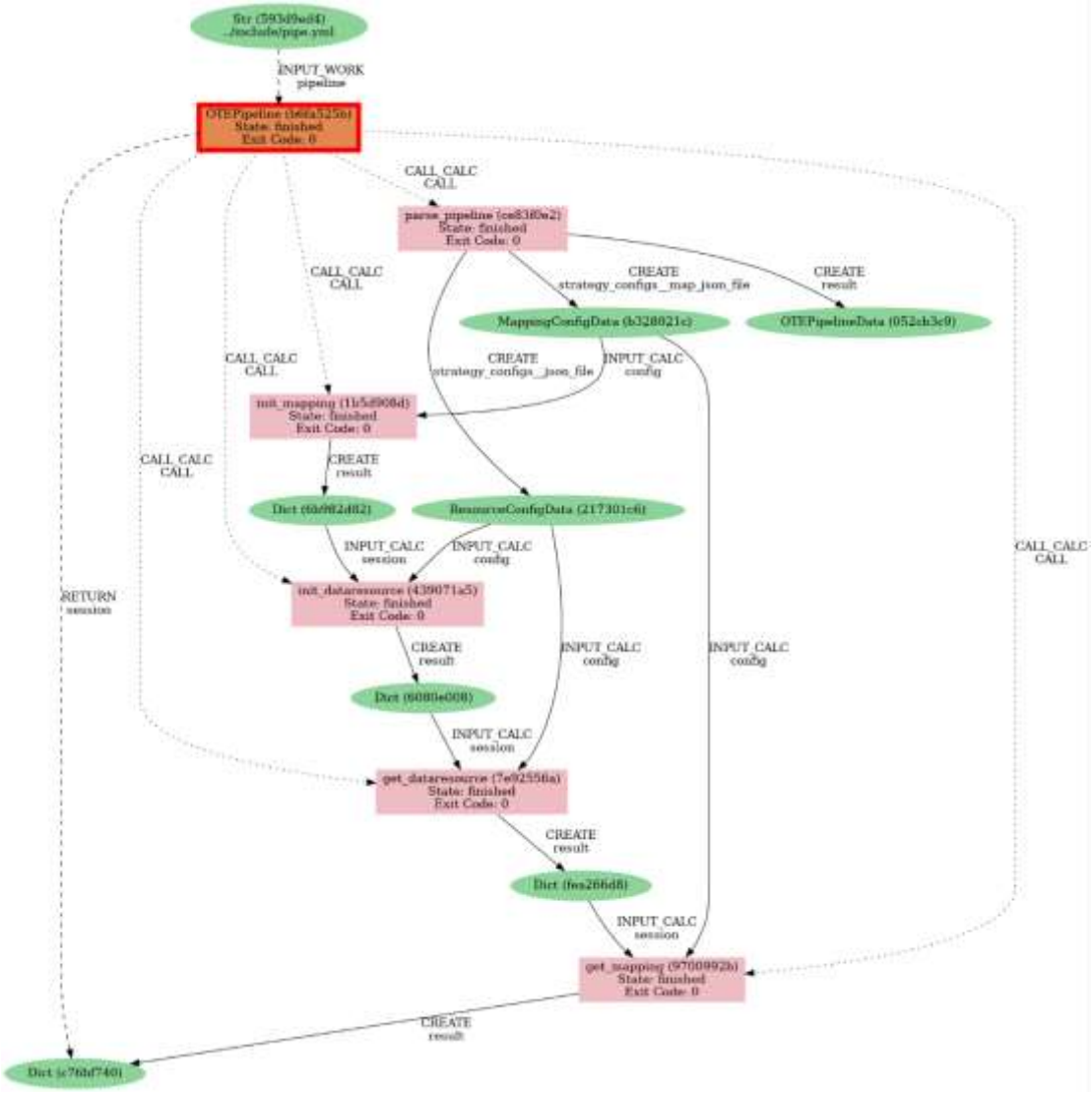
To execute the **DeclarativePipeline** in AiiDA, import the **aiida.engine.run_get_pk** function. This function will allow for a synchronous, blocking execution of a process (AiiDA will wait until the process has finished). We also need to import the **aiida.plugins.DataFactory** and **aiida.plugins.WorkflowFactory** classes. The process executes with the workflow entrypoint *"execflow.pipeline"* with the **DeclarativePipeline** (encapsulated in an AiiDA Data Node) as argument.

```python
from aiida.engine import run_get_pk
from aiida.plugins import DataFactory, WorkflowFactory

path_to_yaml_file = DataFactory("core.str")("../include/pipe.yml")

result, workflow_pk = run_get_pk(
    WorkflowFactory("execflow.pipeline"),
    pipeline=path_to_yaml_file,
)
```

By executing this function, an AiiDA workflow will execute OTEAPI strategies defined in the **DeclarativePipeline** file. A provenance graph from the pipeline execution can be generated using **verdi**, a command-line interface (CLI) tool for interacting with AiiDA.



The overall hierarchy employed by OpenModel for running ExecFlow and the associated **DeclarativeChain** and **DeclarativePipeline** elements, will be to execute a single **DeclarativeChain**, which may then *call* **DeclarativePipeline**s to both retrieve data used as input for calculations run in AiiDA and semantically map and parse eventual outputs. All files will be generated by OntoFlow. This allows integration of materials simulation and modelling software across several levels of integration, as well full semantic interoperability.

4 ACKNOWLEDGMENT

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 953167.