



Cite this: *Digital Discovery*, 2025, **4**, 3149

A python workflow definition for computational materials design

Jan Janssen, ^{*a} Janine George, Julian Geiger, ^d Marnik Bercx, Xing Wang, ^d Christina Ertural, ^b Jörg Schaarschmidt, ^e Alex M. Ganose, Giovanni Pizzi, ^d Tilmann Hückel ^{ab} and Jörg Neugebauer

Numerous Workflow Management Systems (WfMS) have been developed in the field of computational materials science with different workflow formats, hindering interoperability and reproducibility of workflows in the field. To address this challenge, we introduce here the Python Workflow Definition (PWD) as a workflow exchange format to share workflows between Python-based WfMS, currently AiiDA, jobflow, and pyiron. This development is motivated by the similarity of these three Python-based WfMS, that represent the different workflow steps and data transferred between them as nodes and edges in a graph. With the PWD, we aim at fostering the interoperability and reproducibility between the different WfMS in the context of Findable, Accessible, Interoperable, Reusable (FAIR) workflows. To separate the scientific from the technical complexity, the PWD consists of three components: (1) a conda environment that specifies the software dependencies, (2) a Python module that contains the Python functions represented as nodes in the workflow graph, and (3) a workflow graph stored in the JavaScript Object Notation (JSON). The first version of the PWD supports Directed Acyclic Graph (DAG)-based workflows. Thus, any DAG-based workflow defined in one of the three WfMS can be exported to the PWD and afterwards imported from the PWD to one of the other WfMS. After the import, the input parameters of the workflow can be adjusted and computing resources can be assigned to the workflow, before it is executed with the selected WfMS. This import from and export to the PWD is enabled by the PWD Python library that implements the PWD in AiiDA, jobflow, and pyiron.

Received 26th May 2025
Accepted 1st October 2025

DOI: 10.1039/d5dd00231a
rsc.li/digitaldiscovery

1 Introduction

Due to their intrinsic hierarchical nature, material properties depend on the coupling of various domains, among others, materials chemistry, defect engineering, microstructure physics, and mechanical engineering. This often requires multiscale simulation approaches to adequately model materials with different communities representing the different scales. Consequently, the goal of multiscale simulations in materials science is to bridge the gap between the macroscale relevant for applying these materials and the quantum mechanical *ab initio* approach of a universal parameter-free description of materials at the atomic scale. One of these multiscale simulation approaches that has recently gained popularity is coupling the electronic-structure scale and atomic scale by training machine-learned interatomic potentials (MLIP).¹ Such a training of a MLIP

typically consists of the generation of a reference dataset of electronic structure simulations, the fitting of the MLIP with a specialized fitting code, typically written in Python based on machine learning frameworks like pytorch and tensorflow, and the validation of the MLIP with atomistic simulations, often with widespread software such as the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS)² or the atomic simulation environment (ASE),³ both of which also provide Python interfaces. Consequently, it requires expertise in electronic structure simulations, in fitting the MLIP, as well as in interatomic potential simulation, with the corresponding simulation and fitting codes being developed by different communities.^{4,5} The resulting challenge of managing simulation codes from different communities in a combined study of hundreds or thousands of simulations has led to the development of a number of Workflow Management Systems (WfMS). Similarly, high-throughput screening studies, which also couple large numbers of simulations executed with simulation codes at different scales, with different computational costs, and developed from different communities, benefit from WfMS.

In this context, a scientific workflow is commonly defined as the reproducible protocol of a series of process steps, including the transfer of information between them.^{6,7} This can be

^aMax Planck Institute for Sustainable Materials, 40237 Düsseldorf, Germany. E-mail: janssen@mpi-susmat.de

^bBundesanstalt für Materialforschung und -prüfung, 12205 Berlin, Germany

^cFriedrich-Schiller-Universität Jena, 07743 Jena, Germany

^dPSI Center for Scientific Computing, Theory and Data, 5232 Villigen PSI, Switzerland

^eKarlsruhe Institute of Technology (KIT), 76344 Eggenstein-Leopoldshafen, Germany

^fImperial College London, 80 Wood Lane, W12 7TA London, UK



visualized as a graph with the nodes referencing the computational tools and the edges the information transferred between those nodes. Correspondingly, a WfMS is a software tool to orchestrate the construction, management, and execution of the workflow.⁸ The advantages of using a WfMS are: (1) Automated execution of the workflow nodes on high-performance computing (HPC) clusters; (2) improved reproducibility, documentation, and distribution of workflows based on a standardized format; (3) user-friendly interface for creating, editing, and executing workflows; (4) interoperability of scientific software codes; (5) orchestration of high-throughput studies with a large number of individual calculations; (6) out-of-process caching of the data transferred *via* the edges of the workflow and storage of the final results; (7) interfaces to community databases for accessing and publishing data.⁷ As a consequence, using a WfMS abstracts the technical complexity, and the workflow centers around the scientific complexity.

In contrast to WfMS in other communities like BioPipe,⁹ which defines workflows in the Extensible Markup Language (XML), or SnakeMake,¹⁰ NextFlow¹¹ and Common Workflow Language (CWL),¹² which introduce their own workflow languages, many WfMS in the computational materials science community use Python as the workflow language.^{13–24} Using a programming language to define workflows has the benefit that flow control elements, like loops and conditionals, are readily available as basic features of the language, which is not the case for static languages. This is a limitation of static languages, such as XML (more on this in Sec. 1 and the SI). Furthermore, the choice of Python in the field of computational materials science has three additional advantages: (1) the Python programming language is easy to learn as its syntax is characterized by very few rules and special cases, resulting in better readability compared to most workflow languages and a large number of users in the scientific community, (2) the improved computational efficiency of transferring large amounts of small data objects between the different workflow steps in-memory, compared to file-based input and output (IO), and (3) a large number of scientific libraries for the Python programming language, including many for machine learning, materials science and related domain sciences.

The increasing number of WfMS being developed in the computational materials science community and beyond led to the development of benchmarks implementing the same workflow in different WfMS²⁵ and the extension of the FAIR (Findable, Accessible, Interoperable, and Reusable) principles to FAIR workflows.⁸ However, the interoperability between different WfMS remains challenging, even within the subgroup of WfMS that use Python as the workflow language. For this specific case, three levels of interoperability can be identified: (1) the same scientific Python functions are shared between multiple WfMS, *e.g.*, parsers for the input and output files of a given simulation code, (2) the Python functions representing the nodes and the corresponding edges are shared as a template, so that the same workflow can be executed with multiple WfMS and (3) the workflow template, including the intermediate results of the workflow, *e.g.*, the inputs and outputs of each node, is shared.

In the following, the Python Workflow Definition (PWD) for Directed Acyclic Graphs (DAG) and the corresponding Python interface²⁶ are introduced. They implement the second level of interoperability for the following three WfMS: AiiDA,^{13,14,27} jobflow,¹⁶ and pyiron.²⁰ The interoperability of the PWD is demonstrated in three examples: (1) The coupling of Python functions, (2) the calculation of an energy-*versus*-volume curve with the Quantum ESPRESSO Density Functional Theory (DFT) simulation code^{28,29} and (3) the benchmark file-based workflow for a finite element simulation introduced in ref. 25. These three examples highlight the application of the PWD to pure Python workflows, file-based workflows based on calling external executables with file transfer between them, and mixed workflows that combine Python functions and external executables. Different users have different preferences for their choice of WfMS and the PWD is not intended to replace any of them, instead it is an interoperability format to allow users of different WfMS to exchange their workflows.

2 Python workflow definition

Following the goal of separating technical complexity from scientific complexity, our suggestion for a PWD consists of three parts: (1) The software dependencies of the workflow are specified in a conda environment file, so all dependencies can be installed using the conda package manager, which is commonly used in the scientific community.³⁰ (2) Additional Python functions, which represent the nodes in the workflow graph, are provided in a separate Python module. (3) Finally, the workflow graph with nodes and edges is stored in the JavaScript Object Notation (JSON) with the nomenclature inspired by the Eclipse Layout Kernel (ELK) JSON format.³¹ This is illustrated in Fig. 1, together with the three WfMS currently supporting the PWD. If all the involved scientific functionalities are already available within preexisting conda packages, the Python module (part 2) is not required. Still, while an increasing number of open-source simulation codes and utilities for atomistic simulations are available on conda for different scientific domains,³⁰ in most cases, additional Python functions are required. These functions are typically stored in the Python module.

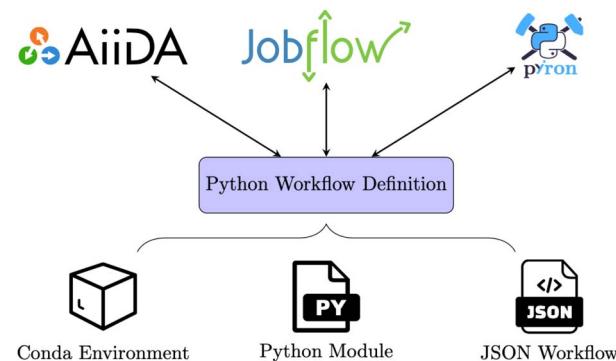


Fig. 1 The Python Workflow Definition (PWD) consists of three components: a conda environment, a Python module, and a JSON workflow representation. The three Workflow Management Systems AiiDA, jobflow, and pyiron all support both importing and exporting to and from the PWD.



As a first simple example workflow, the addition of the product and quotient of two numbers, $c = a/b + a \cdot b$, and subsequent squaring of their sum is represented in the PWD. To illustrate the coupling of multiple Python functions, this computation is split into three Python functions, a `get_prod_and_div()` function to compute the product and quotient of two numbers, a `get_sum()` function for the summation, and a `get_square()` function to raise the number to the power of two:

```

1  def get_prod_and_div(
2      x: float = 1.0, y: float = 1.0
3  ) -> dict[str, float]:
4      return {"prod": x * y, "div": x / y}
5
6  def get_sum(x, y):
7      return x + y
8
9  def get_square(x):
10     return x**2

```

It is important to note here, that the Python functions are defined independently of a specific WfMS, so they can be reused with any WfMS or even without. Furthermore, the Python functions highlight different levels of complexity supported by the PWD: The `get_prod_and_div()` function returns a dictionary with two output variables, with the keys `"prod"` and `"div"` referencing the product and quotient of the two input parameters. Instead, the summation function `get_sum()` takes two input variables and returns only a single output, which is then fed into the `get_square()` function that returns the final result. In addition, the `get_prod_and_div()` function uses default parameter values and type hints, which are optional features of the Python programming language supported by the PWD to improve the interoperability of the workflow. While the computation of the product and quotient of two numbers could be done in two separate functions, the purpose here is to demonstrate the implementation of a function with more than one return value. Another example of such a function could be a matrix diagonalization function that returns the eigenvalues and eigenvectors. The supplementary information provides a more in-depth discussion of how function returns are resolved to an unambiguous mapping in the graph.

As a demonstration, the Python functions `get_prod_and_div()`, `get_sum()` and `get_square()` are stored in a Python module named `workflow.py`. In addition, as these functions have no dependencies other than the Python standard library, the conda environment, `environment.yml`, is sufficiently defined by specifying the Python version:

```

1  channels:
2  - conda-forge
3  dependencies:
4  - python=3.12

```

The `conda-forge` community channel is selected as the package source as it is freely available and provides a large number of software packages for materials science and related disciplines.³⁰ For other examples, *e.g.*, the calculation of the

energy-*versus*-volume curve with `Quantum ESPRESSO` (see below), the conda environment would contain the software dependencies of the workflow, including the simulation code and additional utilities like parsers. It is important to note that the combination of the Python module and the conda environment already addresses the requirements for the first level of interoperability defined above. As the scientific Python functions are defined independently of any workflow environment, they can be used with any WfMS that supports Python functions as nodes. Furthermore, conda environments can be converted to containers, such as docker³² and PyPI packages can be converted to conda packages,³³ highlighting the interoperability advantage of using conda packages.

The limitation of the first level of interoperability is the loss of connection of the individual functions, that is, which output of one function is reused as input of another function. In terms of the workflow as a graph with the Python functions representing the nodes of the graph, these connections are the edges between the nodes. To define the workflow, we wrap the individual function calls in another function to which we can then pass our input values and from which we retrieve our output value:

```

1  def workflow(x: float = 1, y: float = 2):
2      tmp_dict = get_prod_and_div(x=x, y=y)
3      tmp_sum = get_sum(
4          x=tmp_dict["prod"],
5          y=tmp_dict["div"],
6      )
7      return get_square(x=tmp_sum)
8
9  result = workflow(x=1, y=2)

```

We pass the inputs `x=1.0` and `y=2.0` to our `workflow` function, in which the computation of the product and quotient with the `get_prod_and_div()` is executed first. This is then followed by a summation of the two results with the `get_sum()` function, which returns a single output value that is then fed into the `get_square()` function. The corresponding graph is visualized in Fig. 2.

In the next step, the resulting graph is serialized to an internal JSON representation with the nomenclature and overall structure inspired by the ELK JSON format,³¹ for sharing the workflow between different WfMS. While human-readable, the JSON format is not intended for direct user interaction, *i.e.* generating or modifying the JSON with a text editor; rather, it is primarily focused on enabling interoperability of WfMS and long-term storage. For the construction of a workflow, we recommended using one of the existing WfMS and afterwards exporting the workflow to the PWD. The resulting PWD JSON for the arithmetic workflow is:

On the first level, the PWD JSON format defines the workflow metadata given by the version number, nodes and edges:

- The version number (of the PWD JSON format) is given by three non-negative integers combined in a string, to enable semantic versioning. Minor changes and patches which do not affect the backwards compatibility are indicated by increasing the second and third numbers, respectively. In contrast, an

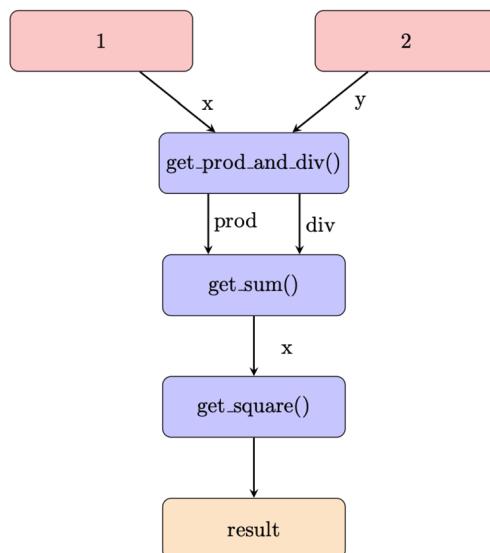


Fig. 2 The arithmetic workflow computes the sum of the product and quotient of two numbers. The red nodes of the workflow graph denote inputs, the orange the outputs, and the blue nodes the Python functions for the computations. The labels of the edges denote the data transferred between the nodes.

```

1  {
2    "version": "1.0.0",
3    "nodes": [
4      {"id": 0, "type": "function",
5       "value": "workflow.get_prod_and_div"}, 
6      {"id": 1, "type": "function",
7       "value": "workflow.get_sum"}, 
8      {"id": 2, "type": "function",
9       "value": "workflow.get_square"}, 
10     {"id": 3, "type": "input",
11      "value": 1, "name": "x"}, 
12     {"id": 4, "type": "input",
13      "value": 2, "name": "y"}, 
14     {"id": 5, "type": "output",
15      "name": "result"} 
16   ],
17   "edges": [
18     {"source": 3, "sourcePort": null,
19      "target": 0, "targetPort": "x"}, 
20     {"source": 4, "sourcePort": null,
21      "target": 0, "targetPort": "y"}, 
22     {"source": 0, "sourcePort": "prod",
23      "target": 1, "targetPort": "x"}, 
24     {"source": 0, "sourcePort": "div",
25      "target": 1, "targetPort": "y"}, 
26     {"source": 1, "sourcePort": null,
27      "target": 2, "targetPort": "x"}, 
28     {"source": 2, "sourcePort": null,
29      "target": 5, "targetPort": null} 
30   ]
31 }
```

increase in the first number indicates changes that are no longer backwards compatible.

- The nodes section is (in this example) a list of six items: The three Python functions defined in the `workflow.py` Python module, the two input parameters for the workflow, in this case $x=1.0$ and $y=2.0$, and the output data node. Each node is defined as a dictionary consisting of an `"id"`, a `"type"`, and a `"value"`. In case of the `"input"` and `"output"` data nodes, the `"name"` is an identifier that denotes how the inputs and outputs are exposed by the overall workflow. Moreover, for `"input"` data nodes, the `"value"` is an optional default value (if provided during workflow construction). On the other hand, for `"function"` nodes, the `"value"` entry contains the module and function name. The usage of the dictionary format allows future extensions by adding additional keys to the dictionary for each node.

- In analogy to the nodes, also the edges are stored as a list of dictionaries. The first two edges connect the input parameters with the `get_prod_and_div()` function. Each edge is defined based on the source node `"source"`, the source port `"sourcePort"`, the target node `"target"` and the target port `"targetPort"`. As the input data nodes do not have associated ports, their source ports are null. In contrast, the target ports are the input parameters x and y of the `get_prod_and_div()` function. The PWD JSON representation also contains two edges that connect the two outputs from the `get_prod_and_div()` function to the inputs of the `get_sum()` function. In analogy to the target port, the source port specifies the output dictionary key to select from the output. If no source port is available (typically because a function does not return a dictionary containing keys that can serve as source ports), then the source port is set to `null` and, in that case, the entire return value of the function (possibly, also a tuple, list, dictionary or any other Python data type) is transferred to the target node. This is the case for the fifth edge that maps the return value of the `get_sum()` function to the `"x"` input of the `get_square()` function. Finally, its result is exposed as the global `"result"` output of the workflow, the last edge in the graph. As the `get_square()` function does return the value directly, and the target of the edge is an output data node (that does not define a port), both `"targetPort"` and `"sourcePort"` are null in this edge.

By using a list of dictionaries for both the nodes and edges, as well as a dictionary at the first level, the PWD JSON format is extensible, and additional metadata beyond the version number can be added in the future. As the focus of this first version of the PWD is the interoperability between the different WfMS, apart from the node types (useful for parsing and validation), no additional metadata is included in the PWD JSON format. To assist the users in analyzing the JSON representation of the PWD, the PWD Python interface provides a `plot()` function to visualize the workflow graph. The `plot()` function is introduced in the supplementary material.

3 Export to the Python workflow definition

The focus of the PWD is to enable the interoperability between different WfMS. Thus, it is recommended that users always use



one of the supported WfMS to create the workflow and export it to the PWD using the PWD Python library. Afterwards, the workflow can be imported into a different WfMS, the input parameters can be modified, and computational resources can be assigned before the workflow is executed. In the following, the same workflow introduced above is defined in AiiDA, jobflow, and pyiron. This highlights the similarities between these Python-based WfMS, which all use the Python programming language as their workflow language, with the selection of WfMS being based on the authors' experience. While this section covers the export of the workflow to the WfMS, the import is discussed in the application section below. Finally, interfaces for additional WfMS are planned in the future. Full integration will be achieved with PWD support becoming an integral part of the WfMS itself and the PWD package possibly becoming a dependency.

3.1 AiiDA

The “Automated Interactive Infrastructure and Database for Computational Science” (AiiDA)^{13,14,27} is a WfMS with a strong focus on data provenance and high-throughput performance. AiiDA provides check pointing, caching, and error handling features for dynamic workflows at full data provenance (*via* an SQL database), among other features. While it originated from the field of computational materials science,³⁴ it has recently been extended to several other fields (see *e.g.* the codes supported in the AiiDA plugin registry³⁵) and to experiments.³⁶ In the following code snippets, we will be using the WorkGraph, a recently added and actively developed new AiiDA workflow component.³⁷ The WorkGraph functions like a canvas for workflow creation to which a user can dynamically add Tasks, that is, workflow components (also called “nodes” in a graph-based representation of a workflow), and connect them with Links (the “edges” in the PWD). This approach to workflow creation offers the flexibility of dynamically chaining workflow components together “on-the-fly”, an approach especially crucial for rapid prototyping common in scientific environments. Implementation of the arithmetic workflow is shown in the following snippets. It starts with the import of relevant modules:

We first import the `python_workflow_definition` module, which contains the necessary code to import from and export to the general Python workflow definition. In addition, from the AiiDA core module, we import AiiDA’s Object-Relational Mapper (ORM), as well as the `load_profile` function. The ORM module allows mapping Python data types to the corresponding entries in

```

1 import python_workflow_definition as pwd
2
3 from aiida import orm, load_profile
4 from aiida_workgraph import WorkGraph, task
5
6 from arithmetic_workflow import (
7     get_sum as _get_sum,
8     get_prod_and_div as _get_prod_and_div,
9     get_square as _get_square
10)
11
12 load_profile()

```

AiiDA’s underlying SQL database, and calling the `load_profile` function ensures that an AiiDA profile (necessary for running workflows *via* AiiDA) is loaded. From the `aiida-workgraph` module, we import the main `WorkGraph` class, as well as the `task` decorator. Lastly, we import the Python functions from the `arithmetic_workflow` module.

To convert the pure Python functions from the arithmetic workflow into AiiDA `WorkGraph` workflow components, we wrap them with the `task` function (decorator):

```

1 get_prod_and_div = task(outputs=["prod", "div"])(_
2     _get_prod_and_div
3 )
4 get_sum = task()(_get_sum)
5 get_square = task()(_get_square)

```

As the `get_prod_and_div` function returns a dictionary with multiple outputs, we pass this information to the `task` function *via* the `outputs` argument, such that we can reference them at a later stage (they will become the ports in the PWD JSON). Without the `outputs` argument, the whole output dictionary `{"prod": x * y, "div": x / y}` would be wrapped as one port with the default `"result"` key. This is what actually happens to the single return value of the `get_sum()` function (as further outlined in the supplementary information, we follow a similar approach to resolve the “ports” entries in the “edges” of the PWD). Next follows the instantiation of the `WorkGraph`:

```
1 wg = WorkGraph("arithmetic")
```

Which then allows adding the previously defined Tasks:

```

1 get_prod_and_div_task = wg.add_task(
2     get_prod_and_div,
3     x=orm.Float(1.0),
4     y=orm.Float(2.0),
5 )
6 get_sum_task = wg.add_task(
7     get_sum,
8     x=get_prod_and_div_task.outputs.prod,
9     y=get_prod_and_div_task.outputs.div,
10 )
11 get_square_task = wg.add_task(
12     get_square,
13     x=get_sum_task.outputs.result,
14 )

```

Here, we wrap the inputs as AiiDA ORM nodes to ensure they are registered as nodes when exporting to the PWD. Further, in the `get_sum_task`, the outputs of the previous `get_prod_and_div_task` are passed as inputs. Note that at this stage, the workflow has not been run, and these output values do not exist yet. In `WorkGraph`, such outputs are represented by a `Socket` that serves as a placeholder for future values and already allows linking them to each other in the workflow:



```

1 In [1]: print(get_prod_and_div_task.outputs.prod)
2 Out[1]: SocketAny(name="prod", value=None)

```

Alternatively, adding tasks to the WorkGraph and linking their outputs can also be done in two separate steps, shown below for linking the `get_prod_and_div_task` and `get_sum_task`:

```

1 ...
2 get_sum_task = wg.add_task(
3     get_sum,
4 )
5 wg.add_link(
6     get_prod_and_div_task.outputs.prod,
7     get_sum_task.inputs.x,
8 )
9 wg.add_link(
10    get_prod_and_div_task.outputs.div,
11    get_sum_task.inputs.y,
12 )
13 ...

```

Lastly, the JSON file containing the PWD can be written to disk via:

```

1     pwd.aiida.write_workflow_json(
2         wg=wg,
3         file_name="arithmetic.json"
4     )

```

The import of the workflow that is exported from AiiDA, in jobflow and pyiron is discussed in Sec. 4. In addition, the corresponding examples on GitHub contain both the import and export for each of the three examples, to highlight the interoperability between the different WfMS.²⁶

3.2 Jobflow

Jobflow¹⁶ was developed to simplify the development of high-throughput workflows. It uses a decorator-based approach to define the Job's that can be connected to form complex workflows (Flows). Jobflow is the workflow language of the workflow library atomate²,³⁸ designed to replace atomate,³⁹ which was central to the development of the Materials Project⁴⁰ database.

First, the `job` decorator, which allows the creation of `Job` objects, and the `Flow` class are imported. In addition, the PWD Python module and the functions of the arithmetic workflow are imported in analogy to the previous example.

```

1 from jobflow import job, Flow
2 import python_workflow_definition as pwd
3 from arithmetic_workflow import (
4     get_sum as _get_sum,
5     get_prod_and_div as _get_prod_and_div,
6     get_square as _get_square,
7 )

```

Using the `job` object decorator, the imported functions from the arithmetic workflow are transformed into jobflow `Jobs`. These `Jobs` can delay the execution of Python functions and can be chained into workflows (Flows). A `Job` can return serializable outputs (e.g., a number, a dictionary, or a Pydantic model) or a so-called `Response` object, which enables the execution of dynamic workflows where the number of nodes is not known prior to the workflow's execution. As jobflow itself is only a workflow language, the workflows are typically executed on high-performance computers with a workflow manager such as Fireworks⁴¹ or jobflow-remote.⁴² For smaller and test workflows, simple linear, non-parallel execution of the workflow graph can be performed with jobflow itself. All outputs of individual jobs are saved in a database. For high-throughput applications, typically, a MongoDB database is used. For testing and smaller workflows, a memory database can be used instead. In Fireworks, its predecessor in the Materials Project infrastructure, this option did not exist, which was a significant drawback.

```

1 get_prod_and_div = job(_get_prod_and_div)
2 get_sum = job(_get_sum)
3 get_square = job(_get_square)
4
5 prod_and_div = get_prod_and_div(x=1.0, y=2.0)
6 tmp_sum = get_sum(
7     x=prod_and_div.output.prod,
8     y=prod_and_div.output.div,
9 )
10 result = get_square(x=tmp_sum.output)
11
12 flow = Flow([prod_and_div, tmp_sum, result])

```

As before in the AiiDA example, the workflow has not yet been run. `prod_and_div.output.div` refers to an `OutputReference` object instead of the actual output.

Finally, after the workflow is constructed, it can be exported to the PWD using the PWD Python package to store the jobflow workflow in the JSON format, which again can be imported with AiiDA and pyiron as demonstrated in the examples in the GitHub repository.²⁶

```

1 pwd.jobflow.write_workflow_json(
2     flow=flow,
3     file_name="arithmetic.json",
4 )

```

3.3 pyiron

The pyiron WfMS was developed with a focus on rapid prototyping and up-scaling atomistic simulation workflows.²⁰ It has since been extended to support simulation workflows at different scales, including the recent extension to experimental workflows.⁴³ Based on this generalization, the same arithmetic Python workflow is implemented in the pyiron WfMS. Starting with the import of the pyiron `job` object decorator and the PWD Python module, the functions of the



arithmetic workflow are imported in analogy to the previous examples above.

```

1  from pyiron_base import job
2  import python_workflow_definition as pwd
3  from arithmetic_workflow import (
4      get_sum as _get_sum,
5      get_prod_and_div as _get_prod_and_div,
6      get_square as _get_square,
7  )

```

Using the job object decorator, the imported functions from the arithmetic workflow are converted to pyiron job generators. These job generators can be executed like Python functions; still, internally, they package the Python function and corresponding inputs in a pyiron job object, which enables the execution on HPC clusters by assigning dedicated computing resources and provides the permanent storage of the inputs and output in the Hierarchical Data Format (HDF5). For the `get_prod_and_div()` function, an additional list of output parameter names is provided, which enables the coupling of the functions before the execution, to construct the workflow graph.

```

1  get_sum = job(_get_sum)
2  get_prod_and_div = job(
3      _get_prod_and_div,
4      output_key_lst=["prod", "div"],
5  )
6  get_square = job(_get_square)

```

After the conversion of the Python functions to pyiron job generators, the workflow is constructed. The pyiron job generators are called just like Python functions; still, they return pyiron delayed job objects rather than the computed values. These delayed job objects are linked with each other by using a delayed job object as an input to another pyiron job generator. Finally, the whole workflow would be only executed once the pull function `pull()` is called on the delayed pyiron object of the `get_square()` function. At this point, the delayed pyiron objects are converted to pyiron job objects, which are executed using the pyiron WfMS. In particular, the conversion to pyiron job objects enables the automated caching to the hierarchical data format (HDF5) and the assignment of computing resources.

```

1  prod_and_div = get_prod_and_div(x=1.0, y=2.0)
2  tmp_sum = get_sum(
3      x=prod_and_div.output.prod,
4      y=prod_and_div.output.div,
5  )
6  result = get_square(x=tmp_sum)

```

For the example here, the workflow execution is skipped and the workflow is exported to the PWD using the PWD Python package to store the pyiron workflow in JSON format. The export command is implemented in analogy to the export commands for AiiDA and jobflow, taking a delayed pyiron object as an input in combination with the desired file name for the JSON representation of the workflow graph, which again can be imported

with AiiDA and jobflow as demonstrated in the examples in the GitHub repository.²⁶

```

1  pwd.pyiron_base.write_workflow_json(
2      delayed_object=result,
3      file_name="arithmetic.json",
4  )

```

The implementation of the arithmetic workflow in pyiron demonstrates the similarities to AiiDA and jobflow.

4 Import from the Python workflow definition

To demonstrate the application of the PWD beyond just the arithmetic example above, we consider a second workflow that describes the calculation of an energy-*versus*-volume curve with Quantum ESPRESSO. The energy-*versus*-volume curve is typically employed to calculate the equilibrium volume and the compressive bulk modulus for bulk materials. The workflow is illustrated in Fig. 3, with the red and orange nodes marking the inputs and outputs of the workflow, the blue nodes the Python functions, and the green nodes indicating Python functions that internally launch Quantum ESPRESSO simulations. The individual steps of the workflow are:

1. Based on the input of the chemical element, the lattice constant, and the crystal symmetry, the atomistic bulk structure is generated by calling the bulk structure generation function `get_bulk_structure()`. This function is obtained *via* the Atomistic Simulation Environment (ASE)³ and extended to enable the serialization of the atomistic structure to the JSON format using the OPTIMADE⁴⁴ Python tools.⁴⁵

2. The structure is relaxed afterwards with Quantum ESPRESSO to get an initial guess for the equilibrium lattice constant. Quantum ESPRESSO is written in FORTRAN and does

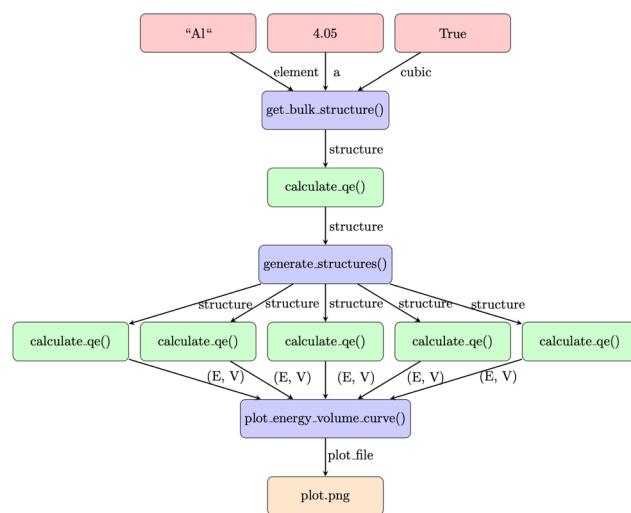


Fig. 3 Energy-*versus*-volume curve calculation workflow with Quantum ESPRESSO. Red boxes denote inputs, orange boxes outputs, blue boxes Python functions and green boxes calls to external executables.



not provide Python bindings, so that the communication is implemented in the `calculate_qe()` function by writing input files, calling the external executable, and parsing the output files. This `calculate_qe()` Python function has been implemented once and is reused by all three WfMS.

3. Following the equilibration, the resulting structure is strained in the function `generate_structures()` with two compressive strains of -10% and -5% and two tensile strains of 5% and 10% . Together with the initially equilibrated structure, this leads to a total of five structures.

4. Each structure is again evaluated with Quantum ESPRESSO to compute the energy of the strained structure.

5. After the evaluation with Quantum ESPRESSO, the calculated energy–volume pairs are collected in the `plot_energy_volume_curve()` function and plotted as an energy-*versus*-volume plot. The final plot is saved in a file named `plot.png`.

Compared to the previous arithmetic example, this workflow is more advanced and not only illustrates one-to-one connections, in terms of one node being connected to another node, but also one-to-many and many-to-one connections. The latter two are crucial to construct the loop over different strains, compute the corresponding volume and energy pairs, and gather the results in two lists, one for the volumes and one for the energies, to simplify plotting. In addition, it highlights the challenge of workflows in computational materials science to couple Python functions for structure generation, modifications, and data aggregation with simulation codes that do not provide Python bindings and require file-based communication. Given the increased complexity of the workflow, the implementation for the individual WfMS is provided in the supplementary material. Instead, the following briefly highlights how the workflow, which was previously stored in the PWD, can be reloaded with the individual frameworks.

Starting with the AiiDA WfMS, the first step is to load the AiiDA profile and import the PWD Python interface. Afterwards, the workflow can be loaded from the JSON representation `qe.json` using the `load_workflow_json()` function. To demonstrate the capability of modifying the workflow parameters before the execution of the (re-)loaded workflow, we then modify the lattice constant of the `get_bulk_structure()` node to 4.05 \AA . Similarly, one could also adapt the element, bulk structure, or strain list input parameters of the workflow. Finally, the workflow is executed by calling the `run()` function of the AiiDA WorkGraph object:

```

1  from aiida import orm, load_profile
2  import python_workflow_definition as pwd
3
4  load_profile()
5
6  wg = pwd.aiida.load_workflow_json(
7      file_name="qe.json"
8  )
9  wg.tasks[0].inputs.a.value = orm.Float(4.05)
10 wg.run()

```

The same JSON representation `qe.json` of the workflow can also be loaded with the jobflow WfMS. Again, the jobflow WfMS and the PWD Python interface are imported. The JSON representation `qe.json` is loaded with the `load_workflow_json()` function. Afterwards, the lattice constant is adjusted to 4.05 \AA and finally the workflow is executed with the jobflow `run_locally()` function. We note that the same workflow could also be submitted to a HPC cluster, but local execution is primarily chosen here for demonstration purposes to enable the local execution of the provided code examples.

```

1  from jobflow.managers.local import run_locally
2  import python_workflow_definition as pwd
3
4  flow = pwd.jobflow.load_workflow_json(
5      file_name="qe.json"
6  )
7  flow[0].function_kwargs["a"] = 4.05
8  run_locally(flow)

```

In analogy to the AiiDA WfMS and the jobflow WfMS, the energy-*versus*-volume curve workflow can also be executed with the pyiron WfMS. Starting with the import of the PWD Python interface, the JSON representation `qe.json` of the workflow is again loaded with the `load_workflow_json()` function, followed by the adjustment of the lattice constant to 4.05 \AA by accessing the input of the first delayed job object. Finally, the last delayed job object's `pull()` function is called to execute the workflow.

```

1  import python_workflow_definition as pwd
2
3  wf = pwd.pyiron_base.load_workflow_json(
4      file_name="qe.json"
5  )
6  wf[0].input["a"] = 4.05
7  wf[-1].pull()

```

The focus of this second example is to highlight that a workflow stored in the PWD can be executed with all three workflow frameworks with minimally adjusted code. This not only applies to simple workflows consisting of multiple Python functions but also includes more complex logical structures like the one-to-many and many-to-one connections, covering any Directed Acyclic Graphs (DAG) topology. We remark, though, that in the current version the restriction to DAGs is also a limitation of the PWD, as it does not cover dynamic workflows, such as a while loop that adds additional steps until a given condition is fulfilled. Another challenge is the assignment of computational resources, like the assignment of a fixed number of CPU cores, as the wide variety of different HPC clusters with different availability of computing resources hinders standardization. As such, the user is required to adjust the computational resources *via* the WfMS after reloading the workflow graph. For this reason, the workflow is also not directly executed by the `load_workflow_json()` function, but rather the user can explore and modify the workflow and



afterwards initiate the execution with any of the WfMS once the required computational resources are assigned.

5 Compatibility to non-Python-based workflows

The two previous examples demonstrated Python-based workflows, which couple either solely Python functions or Python functions and external executables, wrapped by other Python functions that write the input files and parse the output files. Before Python-based WfMS, a number of previous WfMS were introduced, which couple simulation codes solely based on transferring files between the different steps of the workflow.⁹⁻¹² To demonstrate that the PWD can also be applied to these file-based workflows, we implement the benchmark published in ref. 25 for file-based workflows in materials science in the PWD. The corresponding workflow is illustrated in Fig. 4.

As the file-based workflow for finite element simulations is already discussed in the corresponding publication,²⁵ it is only summarized here. A mesh is generated in the first pre-processing step, followed by the conversion of the mesh format in the second pre-processing step. Afterwards, the Poisson solver of the finite element code is invoked. Finally, in the postprocessing, the data is first visualized in a line plot, a TeX macro is generated, and a TeX document is compiled, resulting in the paper as the final output. To represent this file-based workflow in the PWD, each node is represented by

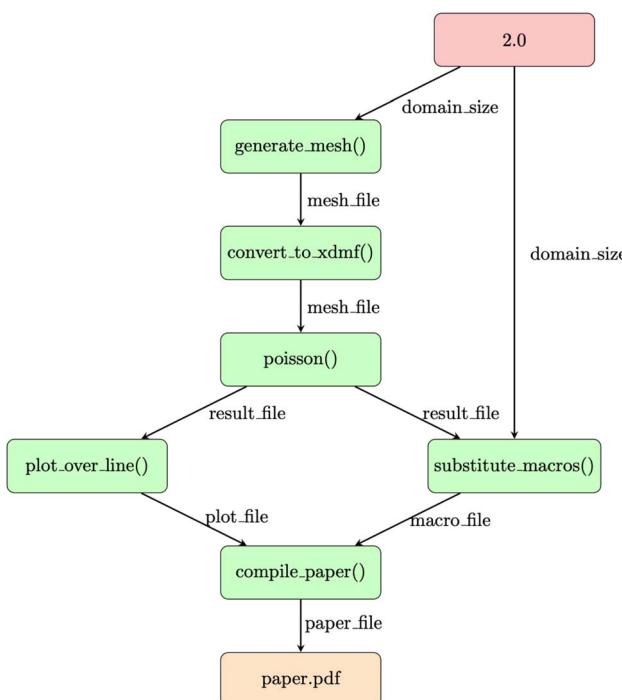


Fig. 4 File-based finite element workflow from ref. 25 implemented with the Python Workflow Definition (PWD). Red nodes denote inputs, orange nodes outputs, green nodes calls to external executables, and the labels on the edges the files and data transferred between them. Files are passed as path objects between the individual steps.

a Python function. This Python function acts as an interface to the corresponding command line tool, handling the writing of the input files, calling of the command line tool and the parsing of the output files. In this specific case, which is purely based on external executables, the output files of one node are copied to be used as input files for the next node, and only the path to the corresponding file is transferred in Python. The Python function for the `generate_mesh()` node is given below:

The input parameters of the `generate_mesh()` function are the `domain_size` and the `source_directory` with the `source_directory` referencing the location of additional input files. Following the definition of a number of variables, a directory is created and the source files are copied as templates to this directory. Then the external executable is called. Here we use the `conda_subprocess` package,⁴⁶ which allows us to execute the external executable in a separate conda environment. This was a requirement of the file-based benchmark workflow.²⁵ Finally, the path to the output file `"square.msh"` is returned as result of the Python function.

While the definition of a Python function for each node is an additional overhead, it is important to emphasize that the Python functions were only defined once, independently of the

```

1  import os
2  from conda_subprocess import check_output
3  import shutil
4
5  def generate_mesh(
6      domain_size: float,
7      source_directory: str
8  ) -> str:
9      stage_name = "preprocessing"
10     output_file_name = "square.msh"
11     source_file_name = "unit_square.geo"
12     os.makedirs(stage_name, exist_ok=True)
13     source_file = os.path.join(
14         source_directory, source_file_name
15     )
16     shutil.copyfile(
17         source_file,
18         os.path.join(stage_name, source_file_name)
19     )
20     _ = check_output(
21         [
22             "gmsh", "-2", "-setnumber",
23             "domain_size", str(domain_size),
24             source_file_name,
25             "-o", output_file_name
26         ],
27         prefix_name=stage_name,
28         cwd=stage_name,
29         universal_newlines=True,
30     )
31     return os.path.abspath(
32         os.path.join(stage_name, output_file_name)
33     )

```

different WfMS and afterwards the same Python functions were used in all three WfMS. Again, the step-by-step implementation in the three different WfMS and the exporting to the PWD is available in the supplementary material. This third example again highlights the universal applicability of the PWD, as it can cover both Python-based workflows and file-based workflows.

Finally, to increase the impact of the PWD and extend its generality beyond the three WfMS discussed in this work, we provide a first proof-of-concept implementation to convert a PWD JSON file to the Common Workflow Language (CWL).¹² In this case each input and output of every node is serialized using the built-in pickle serialization of the Python Standard library. The resulting pickle files are then transferred from one node to another through CWL. To convert a given PWD JSON file, use the `write_workflow()` from the CWL submodule of the PWD Python interface:

```

1 import python_workflow_definition as pwd
2
3 pwd.cwl.write_workflow(
4     file_name="workflow.json"
5 )

```

This Python function creates the corresponding CWL files to represent the individual nodes, as well as the resulting workflow in the CWL, which can then be executed by any CWL engine (given that the necessary dependencies are available on the system). Still, it is important to emphasize that in contrast to the interfaces to the Python-based WfMS, the interface to the CWL is a one-way conversion only from the PWD to the CWL, not the other way around. Furthermore, by converting the workflow to the CWL, the performance benefit of handling the data on the edges of the workflow inside the Python process is lost as the CWL interface is based on file-based communication. Lastly, another notable concept close to the PWD is the graph-based Abstract Syntax Tree (AST)⁴⁷ representation of the Python standard library. For brevity this comparison is discussed in the supplementary information.

6 Conclusions

The Python Workflow Definition (PWD) enables users to develop interoperable workflows to fulfill the requirements for Findable, Accessible, Interoperable and Reusable (FAIR) workflows. The first version of the PWD currently supports Directed Acyclic Graphs (DAGs) based workflows and interoperability between the Workflow Management Systems (WfMS) AiiDA, jobflow, and pyiron. It is based on three components: (1) a conda environment that specifies the software dependencies, (2) a Python module that contains the Python functions represented as nodes in the workflow graph, and (3) a workflow graph stored in the JavaScript Object Notation (JSON). The application of the PWD is demonstrated on three different workflows with different combinations of Python functions and external executables, which require interfacing using file-based communication, highlighting the universal applicability of the PWD. With the corresponding Python interface that we

developed, users can export DAG-based workflows from one WfMS to the PWD and then import the PWD representation of the workflow with any of the supported WfMS. After the import of the workflow, the user still has the option to adjust the input parameters of the workflow, adjust and add WfMS specific features, and assign computational resources to leverage HPC during the execution of the workflow. In the current version, the assignment of the computational environment is not included in the PWD as it is not expected that a user would use multiple WfMS on the same HPC cluster, but rather uses the PWD when transferring a workflow from one HPC cluster with a specific WfMS to a different HPC cluster with a different WfMS. In this case, the assignment of the compute environment changes based on the different HPC resources.

Future development directions of the PWD will focus on broadening its adoption and enhancing its capabilities:

- Engage a wider array of WfMS developers and scientific communities in the joint effort. It is recommended to start with the implementation of the `read_workflow_json()` function, *i.e.*, with the mapping of the workflow stored in the JSON file to the internal workflow data structure of a given WfMS. The inverse, the implementation of the `write_workflow_json()` function, then follows analogously.
- Enable connections to data handling frameworks like dataractor,⁴⁸ and leverage the PWD to create containerized, portable versions of generalized workflows for both simulation and experiment.
- Extend the PWD format to include standardized specifications for submitting workflows to standardized HPC resources, thereby simplifying execution across different infrastructures.
- Transcend PWD's current limitation to DAGs by incorporating support for dynamic flow control elements like loops and conditional branching, enabling the representation of more complex scientific workflows. Ultimately, the vision is to evolve the PWD towards a comprehensive schema capable of capturing all information necessary to define computational workflows, from initial setup to final results, beyond the field of materials science. For this vision the key difference of the PWD in comparison to other workflow standardization efforts is the use of the Python programming language to define workflow nodes, which benefits from the wide adoption of the Python programming language in the scientific community and the direct transfer of data in memory, without requiring to store intermediate results in files.

Author contributions

Jan Janssen: Writing – original draft, conceptualization, investigation, methodology, software, visualization, project administration. Janine George: Writing – original draft, methodology, funding acquisition. Julian Geiger: Writing – original draft, investigation, software. Marnik Bercx: Writing – review & editing, methodology. Xing Wang: Writing – review & editing, investigation, software. Christina Ertural: Writing – review & editing. Jörg Schaarschmidt: Writing – review & editing. Alex Ganose: Writing – review & editing. Giovanni Pizzi: Writing –



review & editing, methodology, funding acquisition. Tilmann Hickel: Writing – review & editing, funding acquisition. Jörg Neugebauer: Writing – review & editing, methodology, funding acquisition.

Conflicts of interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The Python implementation of the Python Workflow Definition `python_workflow_definition` including all the examples from the paper are available at: <https://github.com/pythonworkflow/python-workflow-definition>.²⁶ The same repository is also published on Zenodo: <https://doi.org/10.5281/zenodo.15516180>.⁴⁹

Supplementary information is available. See DOI: <https://doi.org/10.1039/d5dd00231a>.

Acknowledgements

JJ, JS, TH, and JN thank the German Federal Ministry of Education and Research (BMBF) for financial support of the project Innovation-Platform MaterialDigital (<https://www.materialdigital.de>) through project funding FKZ no: 13XP5094A, 13XP5094C, and 13XP5094E. Further JJ, TH and JN also acknowledge funding from the Deutsche Forschungsgemeinschaft (DFG) through the CRC1394 “Structural and Chemical Atomic Complexity – From Defect Phase Diagrams to Material Properties”, project ID 409476157 and the consortium NFDI-MatWerk under the National Research Data Infrastructure, NFDI 38/1, project ID 460247524. CE and JaG acknowledge the Gauss Centre for Supercomputing e.V. (<https://www.gauss-centre.eu>) for funding workflow-related developments by providing generous computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (<https://www.lrz.de>) (Project pn73da). JaG was supported by ERC Grant MultiBonds (grant agreement no: 101161771; Funded by the European Union. Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.) JuG, MB, XW and GP acknowledge financial support from the NCCR MARVEL, a National Centre of Competence in Research, funded by the Swiss National Science Foundation (grant no: 205602), and from the SwissTwins project, funded by the Swiss State Secretariat for Education, Research and Innovation (SERI). GP acknowledges financial support from the Open Research Data Program of the ETH Board (project “PREMISE”: Open and Reproducible Materials Science Research).

References

- 1 R. Jacobs, D. Morgan, S. Attarian, J. Meng, C. Shen, Z. Wu, C. Y. Xie, J. H. Yang, N. Artrith, B. Blaiszik, G. Ceder, K. Choudhary, G. Csanyi, E. D. Cubuk, B. Deng, R. Drautz, X. Fu, J. Godwin, V. Honavar, O. Isayev, A. Johansson, B. Kozinsky, S. Martiniani, S. P. Ong, I. Poltavsky, K. Schmidt, S. Takamoto, A. P. Thompson, J. Westermayr and B. M. Wood, *Curr. Opin. Solid State Mater. Sci.*, 2025, **35**, 101214.
- 2 A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott and S. J. Plimpton, *Comp. Phys. Comm.*, 2022, **271**, 108171.
- 3 A. Hjorth Larsen, J. Jørgen Mortensen, J. Blomqvist, I. E. Castelli, R. Christensen, M. Dułak, J. Friis, M. N. Groves, B. Hammer, C. Hargas, E. D. Hermes, P. C. Jennings, P. Bjerre Jensen, J. Kermode, J. R. Kitchin, E. Leonhard Kolsbørg, J. Kubal, K. Kaasbørg, S. Lysgaard, J. Bergmann Maronsson, T. Maxson, T. Olsen, L. Pastewka, A. Peterson, C. Rostgaard, J. Schiøtz, O. Schütt, M. Strange, K. S. Thygesen, T. Vegge, L. Vilhelmsen, M. Walter, Z. Zeng and K. W. Jacobsen, *J. Phys.: Condens. Matter*, 2017, **29**, 273002.
- 4 S. Menon, Y. Lysogorskiy, A. L. M. Knoll, N. Leimeroth, M. Poul, M. Qamar, J. Janssen, M. Mrovec, J. Rohrer, K. Albe, J. Behler, R. Drautz and J. Neugebauer, *npj Comput. Mater.*, 2024, **10**, 261.
- 5 Y. Liu, J. D. Morrow, C. Ertural, N. L. Fragapane, J. L. A. Gardner, A. A. Naik, Y. Zhou, J. George and V. L. Deringer, *Nat. Commun.*, 2025, **16**, 7666.
- 6 J. Schaarschmidt, J. Yuan, T. Strunk, I. Kondov, S. P. Huber, G. Pizzi, L. Kahle, F. T. Bölle, I. E. Castelli, T. Vegge, F. Hanke, T. Hickel, J. Neugebauer, C. R. C. Rêgo and W. Wenzel, *Adv. Energy Mater.*, 2022, **12**, 2102638.
- 7 S. Bekemeier, C. R. Caldeira Rêgo, H. L. Mai, U. Saikia, O. Waseda, M. Apel, F. Arendt, A. Aschemann, B. Bayerlein, R. Courant, G. Dziwis, F. Fuchs, U. Giese, K. Junghanns, M. Kamal, L. Koschmieder, S. Leineweber, M. Luger, M. Lukas, J. Maas, J. Mertens, B. Mieller, L. Overmeyer, N. Pirch, J. Reimann, S. Schröck, P. Schulze, J. Schuster, A. Seidel, O. Shchyglo, M. Sierka, F. Silze, S. Stier, M. Tegeler, J. F. Unger, M. Weber, T. Hickel and J. Schaarschmidt, *Adv. Eng. Mater.*, 2025, **27**, 2402149.
- 8 C. de Visser, L. F. Johansson, P. Kulkarni, H. Mei, P. Neerincx, K. Joeri van der Velde, P. Horvatovich, A. J. van Gool, M. A. Swertz, P. A. C. t. Hoen and A. Niehues, *PLoS Comput. Biol.*, 2023, **19**, 1–13.
- 9 S. Hoon, K. K. Ratnapu, J.-m. Chia, B. Kumarasamy, X. Juguang, M. Clamp, A. Stabenau, S. Potter, L. Clarke and E. Stupka, *Genome Res.*, 2003, **13**, 1904–1915.
- 10 J. Köster and S. Rahmann, *Bioinformatics*, 2012, **28**, 2520–2522.
- 11 P. D. Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo and C. Notredame, *Nat. Biotechnol.*, 2017, **35**, 316–319.



12 M. R. Crusoe, S. Abeln, A. Iosup, P. Amstutz, J. Chilton, N. Tijanić, H. Ménager, S. Soiland-Reyes, B. Gavrilović, C. Goble and T. C. Community, *Commun. ACM*, 2022, **65**, 54–63.

13 G. Pizzi, A. Cepellotti, R. Sabatini, N. Marzari and B. Kozinsky, *Comput. Mater. Sci.*, 2016, **111**, 218–230.

14 S. P. Huber, S. Zoupanos, M. Uhrin, L. Talirz, L. Kahle, R. Häuselmann, D. Gresch, T. Müller, A. V. Yakutovich, C. W. Andersen, F. F. Ramirez, C. S. Adorf, F. Gargiulo, S. Kumbhar, E. Passaro, C. Johnston, A. Merkys, A. Cepellotti, N. Mounet, N. Marzari, B. Kozinsky and G. Pizzi, *Sci. Data*, 2020, **7**, 300.

15 M. Gjerding, T. Skovhus, A. Rasmussen, F. Bertoldo, A. H. Larsen, J. J. Mortensen and K. S. Thygesen, *Comput. Mater. Sci.*, 2021, **199**, 110731.

16 A. S. Rosen, M. Gallant, J. George, J. Riebesell, H. Sahasrabuddhe, J.-X. Shen, M. Wen, M. L. Evans, G. Petretto, D. Waroquiers, G.-M. Rignanese, K. A. Persson, A. Jain and A. M. Ganose, *J. Open Source Softw.*, 2024, **9**, 5995.

17 S. Vandenhante, M. Cools-Ceuppens, S. DeKeyser, T. Verstraelen and V. V. Speybroeck, *npj Comput. Mater.*, 2023, **9**, 19.

18 J. J. Mortensen, M. Gjerding and K. S. Thygesen, *J. Open Source Softw.*, 2020, **5**, 1844.

19 E. Gelžintytė, S. Wengert, T. K. Stenczel, H. H. Heenen, K. Reuter, G. Csányi and N. Bernstein, *J. Chem. Phys.*, 2023, **159**, 124801.

20 J. Janssen, S. Surendralal, Y. Lysogorskiy, M. Todorova, T. Hickel, R. Drautz and J. Neugebauer, *Comput. Mater. Sci.*, 2019, **163**, 24–36.

21 Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. Wozniak, I. Foster, M. Wilde and K. Chard, *28th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC, 2019.

22 C. S. Adorf, P. M. Dodd, V. Ramasubramani and S. C. Glotzer, *Comput. Mater. Sci.*, 2018, **146**, 220–229.

23 B. H. Sjølin, W. S. Hansen, A. A. Morin-Martinez, M. H. Petersen, L. H. Rieger, T. Vegge, J. M. García-Lastra and I. E. Castelli, *Digital Discovery*, 2024, **3**, 1832–1841.

24 F. Zapata, L. Ridder, J. Hidding, C. R. Jacob, I. Infante and L. Visscher, *J. Chem. Inf. Model.*, 2019, **59**, 3191–3197.

25 P. Diercks, D. Gläser, O. Lünsdorf, M. Selzer, B. Flemisch and J. F. Unger, *ing.grid*, 2023, **1**(1), DOI: [10.48694/inggrid.3726](https://doi.org/10.48694/inggrid.3726).

26 Python Workflow Definition, <https://github.com/pythonworkflow/python-workflow-definition>, accessed: 2025-05-21.

27 M. Uhrin, S. P. Huber, J. Yu, N. Marzari and G. Pizzi, *Comput. Mater. Sci.*, 2021, **187**, 110086.

28 P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni, I. Dabo, A. Dal Corso, S. de Gironcoli, S. Fabris, G. Fratesi, R. Gebauer, U. Gerstmann, C. Gougoussis, A. Kokalj, M. Lazzeri, L. Martin-Samos, N. Marzari, F. Mauri, R. Mazzarello, S. Paolini, A. Pasquarello, L. Paulatto, C. Sbraccia, S. Scandolo, G. Sclauzero, A. P. Seitsonen, A. Smogunov, P. Umari and R. M. Wentzcovitch, *J. Phys.: Condens. Matter*, 2009, **21**, 395502.

29 P. Giannozzi, O. Andreussi, T. Brumme, O. Bunau, M. Buongiorno Nardelli, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, M. Cococcioni, N. Colonna, I. Carnimeo, A. Dal Corso, S. de Gironcoli, P. Delugas, R. A. DiStasio, A. Ferretti, A. Floris, G. Fratesi, G. Fugallo, R. Gebauer, U. Gerstmann, F. Giustino, T. Gorni, J. Jia, M. Kawamura, H.-Y. Ko, A. Kokalj, E. Küçükbenli, M. Lazzeri, M. Marsili, N. Marzari, F. Mauri, N. L. Nguyen, H.-V. Nguyen, A. Otero-de-la Roza, L. Paulatto, S. Poncé, D. Rocca, R. Sabatini, B. Santra, M. Schlipf, A. P. Seitsonen, A. Smogunov, I. Timrov, T. Thonhauser, P. Umari, N. Vast, X. Wu and S. Baroni, *J. Phys.: Condens. Matter*, 2017, **29**, 465901.

30 B. Grüning, R. Dale, A. Sjödin, B. A. Chapman, J. Rowe, C. H. Tomkins-Tinch, R. Valieris, J. Köster and T. B. Team, *Nat. Methods*, 2018, 475–476.

31 Eclipse Layout Kernel JSON Format, <https://eclipse.dev/elk/documentation/tooldevelopers/graphdatastructure/jsonformat.html>, accessed: 2025-05-21.

32 repo2docker, [https://github.com/jupyterhub\(repo2docker\)](https://github.com/jupyterhub(repo2docker)), accessed: 2025-08-21.

33 grayskull, <https://github.com/conda/grayskull>, accessed: 2025-08-21.

34 S. P. Huber, E. Bosoni, M. Bercx, J. Bröder, A. Degomme, V. Dikan, K. Eimre, E. Flage-Larsen, A. Garcia, L. Genovese, D. Gresch, C. Johnston, G. Petretto, S. Poncé, G.-M. Rignanese, C. J. Sewell, B. Smit, V. Tseplyaev, M. Uhrin, D. Wortmann, A. V. Yakutovich, A. Zadoks, P. Zarabadi-Poor, B. Zhu, N. Marzari and G. Pizzi, *npj Comput. Mater.*, 2021, **7**, 136.

35 AiiDA plugin registry, https://aiidateam.github.io/aiida-registry, accessed: 2025-05-21.

36 P. Kraus, E. Bainglass, F. F. Ramirez, E. Svaluto-Ferro, L. Ercole, B. Kunz, S. P. Huber, N. Plainpan, N. Marzari, C. Battaglia and G. Pizzi, *J. Mater. Chem. A*, 2024, **12**, 10773–10783.

37 AiiDA workgraph documentation, https://aiida-workgraph.readthedocs.io/en/latest, accessed: 2025-05-22.

38 A. M. Ganose, H. Sahasrabuddhe, M. Asta, K. Beck, T. Biswas, A. Bonkowski, J. Bustamante, X. Chen, Y. Chiang, D. C. Chrzan, J. Clary, O. A. Cohen, C. Ertural, M. Gallant, J. George, S. Gerits, R. E. A. Goodall, R. Guha, G. Hautier, M. Horton, A. D. Kaplan, R. Kingsbury, M. C. Kuner, B. Li, X. Linn, M. McDermott, R. S. Mohanakrishnan, A. N. Naik, J. B. Neaton, K. A. Persson, G. Petretto, T. Purcell, F. Ricci, B. Rich, J. Riebesell, G.-M. Rignanese, A. S. Rosen, M. Scheffler, J. Schmidt, J.-X. Shen, A. Sobolev, R. Sundararaman, C. Tezak, V. Trinquet, J. Varley, D. Vigil-Fowler, D. Wang, D. Waroquiers, M. Wen, H. Yang, H. Zheng, J. Zheng, Z. Zhu and A. Jain, *Digital Discovery*, 2025, **4**, 1944–1973.

39 K. Mathew, J. H. Montoya, A. Faghaninia, S. Dwarakanath, M. Aykol, H. Tang, I.-H. Chu, T. Smidt, B. Bocklund, M. Horton, J. Dagdelen, B. Wood, Z.-K. Liu, J. Neaton,

S. P. Ong, K. Persson and A. Jain, *Comput. Mater. Sci.*, 2017, **139**, 140–152.

40 A. Jain, S. P. Ong, G. Hautier, W. Chen, W. D. Richards, S. Dacek, S. Cholia, D. Gunter, D. Skinner, G. Ceder and K. A. Persson, *APL Mater.*, 2013, **1**, 011002.

41 A. Jain, S. P. Ong, W. Chen, B. Medasani, X. Qu, M. Kocher, M. Brafman, G. Petretto, G.-M. Rignanese, G. Hautier, D. Gunter and K. A. Persson, *Concurr. Comput. Pract. Exp.*, 2015, **27**, 5037–5059.

42 G. Petretto, M. Evans, D. Waroquier, F. Ricci, J. Riebesell and C. Ertural, jobflow-remote, 2024, <https://github.com/Matgenix/jobflow-remote/tree/v0.1.4>.

43 M. Stricker, L. Banko, N. Sarazin, N. Siemer, J. Janssen, L. Zhang, J. Neugebauer and A. Ludwig, Computationally accelerated experimental materials characterization – drawing inspiration from high-throughput simulation workflows, 2025, <https://arxiv.org/abs/2212.04804>.

44 M. L. Evans, J. Bergsma, A. Merkys, C. W. Andersen, O. B. Andersson, D. Beltrán, E. Blokhin, T. M. Boland, R. Castañeda Balderas, K. Choudhary, A. Díaz Díaz, R. Domínguez García, H. Eckert, K. Eimre, M. E. Fuentes Montero, A. M. Krajewski, J. J. Mortensen, J. M. Nápoles Duarte, J. Pietryga, J. Qi, F. d. J. Trejo Carrillo, A. Vaitkus, J. Yu, A. Zettel, P. B. de Castro, J. Carlsson, T. F. T. Cerqueira, S. Divilov, H. Hajiyani, F. Hanke, K. Jose, C. Oses, J. Riebesell, J. Schmidt, D. Winston, C. Xie, X. Yang, S. Bonella, S. Botti, S. Curtarolo, C. Draxl, L. E. Fuentes Cobas, A. Hospital, Z.-K. Liu, M. A. L. Marques, N. Marzari, A. J. Morris, S. P. Ong, M. Orozco, K. A. Persson, K. S. Thygesen, C. Wolverton, M. Scheidgen, C. Toher, G. J. Conduit, G. Pizzi, S. Gražulis, G.-M. Rignanese and R. Armiento, *Digital Discovery*, 2024, **3**, 1509–1533.

45 M. L. Evans, C. W. Andersen, S. Dwaraknath, M. Scheidgen, Á. Fekete and D. Winston, *J. Open Source Softw.*, 2021, **6**, 3458.

46 Conda Subprocess Package, https://github.com/pyiron/conda_subprocess, accessed: 2025-05-21.

47 Abstract Syntax Trees, <https://docs.python.org/3/library/ast.html>, accessed: 2025-05-21.

48 M. L. Evans, G.-M. Rignanese, D. Elbert and P. Kraus, *MRS Bull.*, 2025, **50**, 838–845.

49 Zenodo, DOI: [10.5281/zenodo.15516180](https://doi.org/10.5281/zenodo.15516180), accessed: 2025-10-09.

