Computational Materials Science 187 (2021) 110086

Contents lists available at ScienceDirect

Computational Materials Science

journal homepage: www.elsevier.com/locate/commatsci

Workflows in AiiDA: Engineering a high-throughput, event-based engine for robust and modular computational workflows

Martin Uhrin [a,b], Sebastiaan P. Huber [a,*], Jusong Yu [c], Nicola Marzari [a], Giovanni Pizzi [a]

[a] Theory and Simulation of Materials (THEOS), and National Centre for Computational Design and Discovery of Novel Materials (MARVEL), École Polytechnique Fédérale de Lausanne, CH-1015 Lausanne, Switzerland
[b] Department of Energy Conversion and Storage, Technical University of Denmark, Kgs. Lyngby DK-2800, Denmark
[c] Department of Physics, South China University of Technology, Guangzhou 510640, China

ARTICLE INFO

Keywords:
Data management
Database
Data sharing
Provenance
Computational workflows
Event-based
Robust computation
High-throughput

ABSTRACT

Over the last two decades, the field of computational science has seen a dramatic shift towards incorporating high-throughput computation and big-data analysis as fundamental pillars of the scientific discovery process. This has necessitated the development of tools and techniques to deal with the generation, storage and processing of large amounts of data. In this work we present an in-depth look at the workflow engine powering AiiDA, a widely adopted, highly flexible and database-backed informatics infrastructure with an emphasis on data reproducibility. We detail many of the design choices that were made which were informed by several important goals: the ability to scale from running on individual laptops up to high-performance supercomputers, managing jobs with runtimes spanning from fractions of a second to weeks and scaling up to thousands of jobs concurrently, and all this while maximising robustness. In short, AiiDA aims to be a Swiss army knife for high-throughput computational science. As well as the architecture, we outline important API design choices made to give workflow writers a great deal of liberty whilst guiding them towards writing robust and modular workflows, ultimately enabling them to encode their scientific knowledge to the benefit of the wider scientific community.

1. Introduction

As developments in computational power have steadily and tremendously increased over the past few decades, so with them the field of computational science. Digital applications have become increasingly complex and often comprise an intricate combination of scientific computational methods and data analysis. The sequence of steps in these applications is often encoded in a workflow and the need to automate these processes has led to an increase in the number of workflow management systems (WMS's) being developed. A WMS provides the necessary functionality to define and subsequently execute workflows that essentially encode a sequence of data transformations [1]. In recent years, many WMS's have been developed which have greatly simplified and streamlined the creation and analysis of data. However, with these improvements have come new challenges in managing the massive amounts of data that are produced.

The most apparent challenge is finding an efficient method of storing the data itself. Although simple storage approaches can solve the problem of data persistence, they fail to address the question of data reproducibility, which carries particular importance in the scientific method and plays a direct role in making data reusable, according to the FAIR Data Principles [2]. Indeed, the reproducibility of data can only be guaranteed if the provenance of data is treated with the same importance as the data itself, as it is the data's provenance that enables its validation and verification [3–6]. Here, it is critical to realise that not only the data themselves, but also the workflows that create them should be part of the tracked provenance. As a consequence of this observation, efforts are underway to extend the FAIR Data Principles to workflows as well [7]. With the rate of data production made possible by modern technologies, it has become untenable to reconstruct the provenance of data *a posteriori*, calling for tools that automatically record it as it is created. Some WMS's have started addressing this challenge, however, their data and workflow provenance guarantees are often insufficient to be able to retrace the origins of a piece of data or to recreate it.

In this paper, we describe in detail the workflow system of AiiDA [8], an open-source, high-throughput, scalable computational infrastructure for automated reproducible workflows and data provenance, implemented in Python. While the design of AiiDA and its workflow system is

* Corresponding author.
E-mail addresses: martin.uhrin.10@ucl.ac.uk (M. Uhrin), mail@sphuber.net (S.P. Huber).

https://doi.org/10.1016/j.commatsci.2020.110086
Received 21 July 2020; Received in revised form 28 September 2020; Accepted 29 September 2020
Available online 16 November 2020

generic enough to be applicable to any computational, and potentially experimental, scientific domain, its origin and strengths lie in applications that make use of high-performance computing (HPC) systems. Users of these environments are often used to scripting, which is why the workflow engine of AiiDA provides a rich application programming interface (API), unlike the majority of WMS that provide a graphical user interface (GUI), such as Kepler [9], Taverna [10] and Triana [11]. An API provides a more direct and seamless integration of the workflow system with the simulation codes and data analysis tools that it manages and that are typically used on HPC systems.

An additional benefit of an API-based workflow language and engine is that it allows for the definition of *dynamic* workflows, whose exact path is not pre-determined but evolves during execution based on the results of completed steps. In AiiDA, for example, the code that defines a workflow is directly executed by the engine and there is no intermediate translation layer. The majority of WMS's, however, interpret workflows that are defined through a static markup language such as XML in the case of Karajan [12], custom XML derivatives such as Askalon's [13] AGWL [14] or workflow specific standards such as the Common Workflow Language (CWL) [15]. Some of them may provide bindings to programming languages in order to define workflows through an API, such as Pegasus [16], however, this is a mere pre-processing step as the workflows are still converted into a Directed Acyclic Graph (DAG) representation in XML, before being executed by the workflow runner when launched. The big disadvantage of these document based workflow definitions is that they are *static*, in the sense that the exact flow of the workflow needs to be known before it is executed[1]. The mechanism also naturally limits the available programming structures to DAGs or directed cyclic graphs (DCG), if loops are supported by the markup language.

The recent Python-based workflow managers Signac [17] and Parsl [18] have chosen a different approach and rely on implicit dataflow to define and control workflows. In this model, new data operations, bound by data dependencies, are executed as their dependencies are fullfilled by other data becoming available in the workspace. The Fireworks system [19], which supports the definition of workflows through documents in JavaScript Object Notation [20], has made important steps toward supporting *dynamic* workflows. A workflow can insert new steps or spawn additional logical branches while it is running, based on intermediate results produced by previously completed steps. However, while this enables runtime-mutable workflows, specific mutations are limited by the constraints of the custom static JSON markup language through which they are defined. In contrast, workflows in AiiDA are implemented directly in Python and as such have all the dynamic expressiveness of a programming language directly at their disposal, as well as full access to the entire provenance graph with the data that is already stored in the database. This proves to be a very powerful mechanism to deal with, for example, the problem of error handling when running high-throughput simulations.

In the field of materials science specifically, other libraries and frameworks exist that provide advanced workflows with automated handling and recovery of errors encountered in *ab initio* calculations, such as AFLOW [21], Atomate [22], MAST [23] and OQMD [24]. However, all of these are typically only compatible with one specific *ab initio* code (with some expanding support to others), whereas the ecosystem of density functional theory (DFT) features a great variety of popular codes [25]. By tightly coupling the WMS and the workflow implementations themselves to any single or a few codes, interoperability is naturally hamstrung. In stark contrast, the workflow system of AiiDA is completely agnostic of the external software that performs the computation and provides an integrated abstract interface for any simulation code, with built-in support for all major resource managing systems, such as PBSPro [26], SLURM [27], SGE [28] and Torque [29]. Through its flexible plugin system, AiiDA allows any code to be made compatible via plugins, which are registered on the AiiDA registry [30] and can be installed with a single command through the Python package manager `pip` [31].

With these considerations, the workflow system of AiiDA has been designed to satisfy the following criteria. The workflow system should (i) facilitate the definition of fully *dynamic* workflows, (ii) with an interface generic enough to support running arbitrary external codes, (iii) automatically store the full provenance of executed workflows (iv) in a way that makes the data easily queryable while scaling towards exascale applications (v) with an overhead of the provenance storage that does not outweigh the cost of the computational workflows themselves. AiiDA's workflow system can be roughly split into two components: the user interface (or API) that allows users to implement workflows and interact with the provenance graph, and the engine that is responsible for automatically executing those workflows and storing the results. In this paper, we first describe the user interface followed by a technical description of the architecture and implementation of the engine.

2. User interface

One of the main design goals of AiiDA's workflow engine interface is to minimize the restrictions imposed on the developer, while simultaneously providing the tools that enable and stimulate the development of maintainable, self-documenting, robust and modular workflows. When forcing interaction with the process engine through a specific API, the amount of functionality disposable to the user is intrinsically limited. To limit these restrictions to the bare minimum, workflows in AiiDA are written directly in Python and the written code is directly executed by the engine, without a translation layer. In this way, the workflow developer has direct access to the entire AiiDA API and all Python libraries, and is not dependent on a particular feature being exposed through a workflow-specific API. Moreover, in the field of computational science, Python is an abundant, thriving and well-supported language, which means that a lot of existing code will naturally interface with AiiDA's workflow engine without any additional custom development required.

On the other hand, not restricting the methods of workflow implementation could lead to a wide variety of solutions, which almost inevitably render them incompatible and non-modular. To counteract this undesired phenomenon while maintaining full access to the AiiDA API, the workflow engine exposes various tools and constructs to simplify the development of workflows, whose use automatically improves robustness and interoperability. In this section, these constructs will be explained in detail, covering their implementation and interface.

A final crucial consideration relates to the extent to which the maintenance of full data provenance, being the core principle of AiiDA, can be guaranteed by its engine. In giving the user almost unrestricted freedom in designing and developing workflows, perfect provenance cannot be guaranteed. As a compromise, the conditions under which provenance *will* be guaranteed by AiiDA, and conversely, how it will definitely be broken, need to be as simple and clear as possible to the user. The design mantra here is once more to restrict the user as little as possible and allow the breaking of data provenance if the user deems it necessary or justified.

2.1. Process specification

Any entity that can be run by AiiDA's engine is named a process and should be implemented through the class `Process`. A process is defined as a set of logical instructions, implemented in code, that operates on a set of inputs in order to produce certain outputs, with the possibility of premature termination through known failure modes. Each `Process` defines its inputs, outputs and known failure modes through its

[1] Logic and loops can often be used but these need to be represented explicitly which quickly becomes cumbersome.

specification, which is facilitated by the `ProcessSpec` class.

2.1.1. Ports and port namespaces

Before diving into the details of how inputs and outputs are specified for a process through its process specification, we clarify the concept of ports and port namespaces. The inputs and outputs of a process share the common feature of being the gateways, or ports, through which data is ported in and out of the black box of a process. These ports can be further grouped or nested in port namespaces. The concepts of a port and a port namespace in AiiDA are implemented by the `Port` and `PortNamespace` classes, respectively. The `PortNamespace` is simply a container of `Port` instances, and given that it is itself a subclass of `Port`, `PortNamespace` instances can be nested within one another. Since the `PortNamespace` is implemented as a mapping, inserting and addressing members of the container is achieved through key referencing as with any other mapping in Python. Each `Port` instance has the following attributes:

- `valid_type`: a tuple of accepted port value types,
- `validator`: a custom validator function to validate the value passed to the port,
- `default`: an optional default port value,
- `required`: a boolean to indicate whether a port value is required, and,
- `non_db`: a boolean to indicate whether the port requires a database storable type.

In addition to these attributes, the `PortNamespace` has the `dynamic` attribute, which is a boolean to indicate whether the namespace can accept any values for ports that are not explicitly defined. The use case for this concept will become clear in a later section on workchains (see Section 2.2.3). When a `Port` or `PortNamespace` is validated, the validation of each port is called recursively, which includes verifying the type of the values passed with respect to the `valid_type` attribute and calling the `validator` function, if defined. A `PortNamespace` is considered valid if and only if all of the ports nested within it, as well as itself, pass validation.

By default, any input to a process should be a database storable type, as otherwise the provenance of the outputs generated by that process would be lost, violating a core principle of AiiDA. However, there are use cases where this isolated loss of provenance is acceptable and putting an absolute requirement on the storability of all inputs might be too restrictive. For this reason, the `non_db` attribute can be used to mark a port as not storable in the database. That is to say, any input that is passed through a port with `non_db` set to `True` will not be stored and linked as an input to the process node that is automatically created in the provenance graph when AiiDA executes the process. AiiDA makes use of this feature to allow defining various process metadata, such as a label or description, which are then not stored as actual input nodes, but directly as attributes of the process node. A user may also decide to use this feature if they deem a particular input as irrelevant for the provenance.

2.1.2. Inputs and outputs

The inputs and outputs of a process are defined through its specification, as implemented by the `ProcessSpec` class. The `ProcessSpec` class contains two `PortNamespace` instances, accessible through the `inputs` and `outputs` attributes, that contain the input and output ports of the process, respectively. To define a new input or output port for a process, the `ProcessSpec` exposes two convenience methods:

Listing 1: The definition of an input and output port through the process specification.

```
spec = ProcessSpec()
spec.input('a', valid_type = Int, default = Int(2), validator =
    is_positive_integer, required = True)
spec.output('b', valid_type = Float, default = Float(-2.0),
    validator = is_negative_float, required = True)
```

These two method calls result in the creation of an `InputPort`, stored in the `inputs` namespace under the key `a`, and an `OutputPort`, stored in the `outputs` namespace under the key `b`, respectively. New input and output namespaces can be created similarly with the `input_namespace` and `output_namespace` methods, respectively, e.g.:

Listing 2: The definition of an input and output port namespace through the process specification.

```
spec = ProcessSpec()
spec.input_namespace('nested.input.namespace')
spec.output_namespace('some.outputs')
```

The argument passed to the methods is used as the key under which the newly created namespaces is inserted into their respective parent namespace. The period is treated as a special character and is interpreted as a namespace separator. The key `nested.input.namespace` is therefore interpreted as a nested namespace of depth three and the port namespaces are recursively created by the `input_namespace` call.

Note that all these `ProcessSpec` methods are declarative in nature and that they can overwrite the effects of previously executed methods. Consider the following example:

Listing 3: The declarative nature of the process specification allows later declarations to override earlier ones.

```
spec = ProcessSpec()
spec.input('a', valid_type = Int, default = Int(2), validator =
    is_positive_integer, required = True)
spec.input('a', valid_type = Float, default = Float(3.0),
    validator = is_positive_float, required = False)
```

The resulting process specification has a single input port *a* in its inputs namespace that accepts float types, as the preceding directive is overwritten.

2.1.3. Exit codes

In addition to inputs and outputs, the process specification is also used to declare the known failure modes of the process. A common method of communicating a particular failure from a process to its caller is through the use of an exit status. An exit status, modelled on a similar concept found in POSIX processes, is defined as an integer that is returned by all processes. If the integer is zero, it signifies that the process executed correctly; any non-zero value indicates an error that maps onto a known failure mode. This concept is implemented in AiiDA through 'exit codes'. Specifically, the `ProcessSpec` class implements the `exit_code` method that allows one to define an exit code for the corresponding process:

Listing 4: The definition of an exit code, consisting of an integer exit status, a reference label and an exit message, through the process specification.

```
spec = ProcessSpec()
spec.exit_code(418, 'ERROR_I_AM_A_TEAPOT', 'the process
    experienced an identity crisis')
```

This example defines an exit code for the process, with exit status 418 and exit message 'the workchain experienced an identity crisis'. These two values are stored in the DB as attributes of the process. In addition, the string `ERROR_I_AM_A_TEAPOT` is a human-readable unique label that can be conveniently used to reference the exit code instead of using the integer status. A detailed explanation of how exit codes are referenced and used in practice is given in paragraph 2.2.8.

2.2. Process implementations

2.2.1. Calculation functions

To honor the design goal of restricting a workflow developer as little as possible, a solution was sought to turn any regular Python function into a fully AiiDA compliant function. Employing the concept of Python's function decorators, a wrapping function that alters or adds to the behavior of the function it is applied to, the `calcfunction` decorator

was developed. To explain its functionality, consider the following Python functions that add and multiply two numbers, respectively:

Listing 5: Two standard Python functions to add and multiply two numbers, respectively.

```
def add(a, b):
    return a + b

def multiply(a, b):
    return a * b
```

By leveraging the `calcfunction` decorator, these plain Python functions are turned into AiiDA compliant functions with the addition of just a single line:

Listing 6: By decorating the Python function with the `calcfunction` decorator, the plain function is automatically transformed by the engine into an AiiDA process when executed.

```
@calcfunction
def add(a, b):
    return a + b

@calcfunction
def multiply(a, b):
    return a * b
```

When either function is called, the decorator instructs the engine to create a `Process` instance on the fly, representing the function. Python's standard `inspect` module is used to introspect the function's signature, which is used to define the input ports for the process specification as explained in Section 2.1.2. Note that, since the process specification is generated from the function signature, not all the functionality of ports and port namespaces are accessible. For example, due to Python's dynamic typing, the expected type for function arguments is not always specified and therefore the `valid_type` attributes for the generated input ports can be undefined. We are currently working to introduce parsing of type annotations to further augment the specification of the generated `Process` to be able to type check the passed parameters.

By simply calling the functions with database-storable types, the engine automatically takes care of creating the corresponding data provenance in the database. For example, the following execution:

Listing 7: Running a decorated function works just as running a normal Python function, with the only difference being that the input values should be database-storable types.

```
multiply(add(Int(3), Int(4)), Int(5))
```

returns the value 35 and creates a representation of the execution in the provenance graph, as represented in Fig. 1. The execution of the two functions are each represented by a `CalcFunctionNode` in the provenance graph, with the corresponding input and output nodes correctly linked to it.

2.2.2. *Work functions*

Even though a workflow could be encoded by means of a concatenation and/or nesting of `calcfunction` calls, that approach does not really capture the logic of a workflow. For instance, from the created provenance graph, it is impossible to ascertain whether the two consecutively called calculations were part of a single coordinated computation, or if the output of the first was simply used as an input to an otherwise unrelated calculation. To define a workflow that captures this logic, the engine provides the `workfunction` decorator, which is analogous to the `calcfunction`, except its purpose is not to *create* new data out of its inputs, but rather to orchestrate a composition of operations. The example of calling two calculation functions directly, as shown in listing 7, could be rewritten as follows:

Listing 8: By decorating the Python function with the `workfunction` decorator, the plain function is automatically transformed by the engine into an AiiDA workflow when executed and 'call' links are added to the calculation functions it calls.

```
1  @calcfunction
2  def add(a, b):
3      return a + b
4
5  @calcfunction
6  def multiply(a, b):
7      return a * b
8
9  @workfunction
10 def add_multiply(x, y, z):
11     sum = add(x, y)
12     product = multiply(sum, z)
13     return product
14
15 result = add_multiply(Int(1), Int(2), Int(3))
```

Simply calling the decorated work function (line 16 of listing 8) will execute the dynamically generated process while storing a representation of it in the provenance graph as shown in Fig. 2.

The work function is not limited to calling only calculation functions, but can also call other work functions, and these are also linked by `CALL` links as shown in Fig. 2. Therefore, arbitrarily deeply nested workflows can be constructed with these two basic components. However, due to their intentional simplicity, the `calcfunction` and `workfunction` decorator solutions (collectively referred to as process functions) also have inherent shortcomings that render them the incorrect tool for certain situations.

In the two examples provided earlier, the computational work that had to be performed in the function body was trivial. However, more often than not, the opposite is the case in computational science applications. A decorated function constitutes a contiguous block of code and will necessary block the interpreter for the duration of the function execution. This implies that for computationally expensive functions, the interpreter will be blocked from executing anything else for extended periods of time. Additionally, intermediate progress cannot be saved in such a way to allow execution to be resumed later at the same point in the source code. This means that when a function is interrupted at any point, all the work it performed up to that point will be lost (note that while Python coroutines would allow the interpreter to switch to executing other code, it would still be nearly impossible to save and later reconstruct the entire state of the call stack). This scenario is particularly relevant considering that the simulations managed by AiiDA can last weeks, and the user might want to stop or restart the machine where AiiDA runs.

Therefore, for all its simplicity of use, process functions should be used sparingly and conscientiously in the development of workflows. The construct implemented in AiiDA that solves all the weak points of

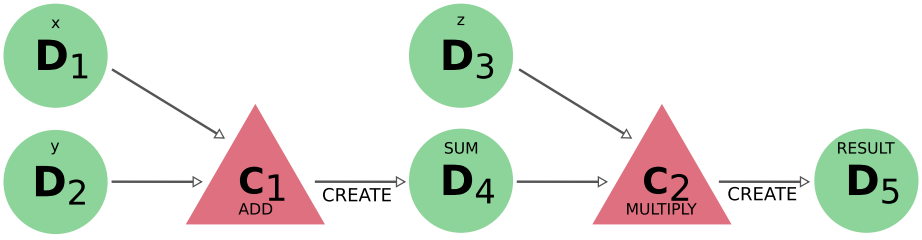

Fig. 1. The provenance that is automatically generated by executing the two calculation functions.

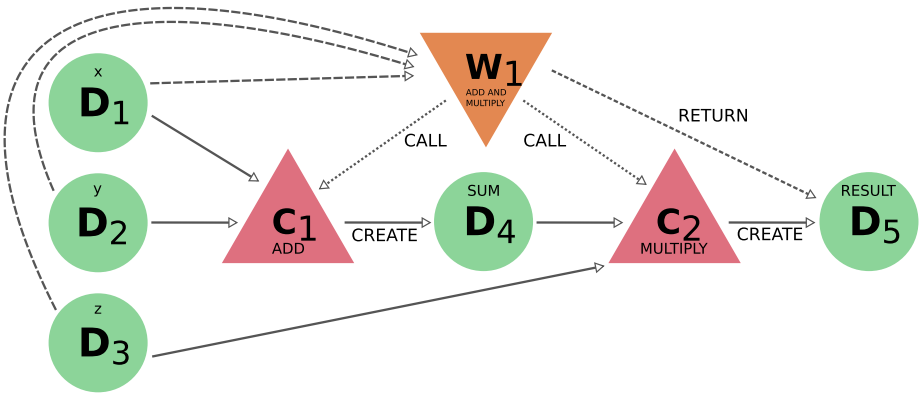

Fig. 2. The provenance graph that is automatically generated by executing the a work function that calls two calculation functions in succession.

the `workfunction` is the `WorkChain`.

2.2.3. *Work chains*

The `WorkChain` class is a subclass of `Process` and is the core component of workflow development in AiiDA. It is used to encode the logic that is typically encompassed by any scientific workflow. The `ProcessSpec` for the `WorkChain` class supports, in addition to the standard properties of the `ProcessSpec` as detailed in Section 2.1, the `outline` method. This method allows a user to define a set of logical rules that, when evaluated, will yield a set of steps that are then executed by the engine. The big advantage of using this construct over a concatenation of work functions is that, in between each step of a work chain, the engine automatically saves the progress to allow a restart from the last checkpoint in the case of execution failure. Additionally, the transition between steps gives the engine the chance to yield the interpreter to other parts of the code, alleviating the blocking behavior inherent in the synchronous execution of the work function construct.

2.2.3.1. Outline. The `outline` method of the work chain's `ProcessSpec` is used to encode the desired logic of a certain workflow. It supports typical logical flow constructs, such as while loops, conditional blocks and return statements. To keep the user interface as simple as possible, the aim was to support a syntax that is as close to standard Python logical constructs as possible. For illustration purposes consider the following description of a trivial workflow. Print the numbers from 0 to 100, replacing the printed number with 'fizz' if it is a multiple of three, with 'buzz' if the number is a multiple of five, and with 'fizzbuzz' if it is both a multiple of three and five. The following listing demonstrates how this logic could be encoded using the syntax of the work chain outline:

Listing 9: An example of a work chain outline that contains logical constructs such as a while loop and conditional statements.

```
spec.outline(
        cls.intialize_to_zero,
        while_(cls.is_less_than_or_equal_to_hundred)(
            if_(cls.is_multiple_of_three_and_five)(
                cls.report_fizz_buzz,
            ).elif_(cls.is_multitple_of_three)(
                cls.report_fizz,
            ).elif_(cls.is_multiple_of_five)(
                cls.report_buzz,
            ).else_(
                cls.report_n,
            ),
            cls.increment_by_one,
        )
)
```

Note that, since the logical constructs `while`, `if`, `elif` and `else` are protected Python keywords, the outline analogues are suffixed with an underscore in order to properly distinguish them from the Python builtins. The statements between or within the calls of the logical constructs represent a step that is to be executed by the engine. Since workflow development in AiiDA happens entirely in Python, the implementation of these steps is achieved by simply implementing them as methods of the `WorkChain` class for which the outline is defined. An example implementation of a subset of these class methods would look something like the following:

Listing 10: Example implementation of some of the outline steps defined in listing 9 as methods of the work chain class.

```
def initialize_to_zero(self):
        self.ctx.n = 0

def is_multiple_of_three(self):
        return self.ctx.n % 3 == 0

...

def report_fizz_buzz(self):
        self.report('fizzbuzz')

...

def increment_by_one(self):
        self.ctx.n += 1
```

Each outline step is defined by a method that takes a single argument `self`, which is a reference to the class instance as is standard in Python. This particularly trivial example already sets up the questions of how log messages can be reported from within a work chain and how data can be passed between outline steps. The answers to these questions will be addressed in the following two sections.

2.2.4. *Reporting*

The `WorkChain` class exposes the `report` method, which takes a single string as an argument, which can be used by the developer to log messages during the execution of the work chain. The `report` method emits the passed log message through the standard Python logging module with a custom log level `REPORT`, which is defined by AiiDA and lies between the `INFO` and `WARNING` log levels. Additionally, the logging configuration of AiiDA defines a database log handler that automatically attaches these logged messages to the node that represents the work chain from which they were emitted. The recorded log messages can then be retrieved through the AiiDA API or its command line interface by referencing the relevant work chain node. This report system is meant for communicating human readable log-like messages as a record of the events that occurred during its execution and should not be parsed to communicate programmatically between processes. For that use case, the concept of exit codes has been implemented, which will be described in greater detail in paragraph 2.2.8.

2.2.5. Checkpoints and context

As mentioned in the introduction of this section, the engine will evaluate the logic defined by the outline and execute the methods that correspond to those outline steps. These methods only take a single argument, `self`, so there is no way to pass data directly from one step to another. To allow data to be passed between the steps of a work chain, each `WorkChain` instance defines a context, which is a simple Python dictionary, accessible through the `ctx` attribute, that is persisted between step transitions. The context can therefore be used as any other Python dictionary to store data and the engine ensures that the state of the work chain instance, along with its context, is saved in a checkpoint in the database (see Section 3.2.1). Through this container, data that was stored in one outline step can be accessed in another.

2.2.6. Calling subprocesses

Work chains can launch any other process as a child process. For example, one can launch a `CalcJob` (a calculation running on an external computer, described in Section 2.2.10) or another `WorkChain` from within a `WorkChain`. The syntax for submitting a process from within a `WorkChain` is identical to submitting a process from a top level Python script, with the exception that one should not use the `submit` free function, but the `submit` method of the `WorkChain` class. For example, to submit a `ChildWorkChain` with a certain set of inputs from within another work chain, one would call:

Listing 11: A subprocess can be submitted through the `submit` method of the `WorkChain` class and the `ToContext` container can be used to register the submitted child process as an awaitable.

```
1 def submit_child_workchain(self):
2         child = self.submit(ChildWorkChain, **inputs)
3         return ToContext(child = child)
```

For the parent work chain to continue, it has to wait for the child process to finish and therefore it has to return control to the interpreter. To communicate to the engine that the work chain needs to wait for a subprocess (in this example the `ChildWorkChain`) to finish, the developer should return an instance of the `ToContext` class. This turns the submitted subprocesses into awaitables, which instructs the engine to halt execution of the work chain until all subprocesses are completed. The same result can be achieved through the `to_context` method of the work chain:

Listing 12: Similar to listing 11, a subprocess can be submitted through the `submit` method of the `WorkChain` class and the `to_context` method can be used to register the submitted process as an awaitable.

```
1 def submit_child_workchain(self):
2         child = self.submit(ChildWorkChain, **inputs)
3         self.to_context(child = child)
```

The engine will proceed to execute the submitted subprocesses and when they are completed, will assign the corresponding process node, used to represent the executed process in the database, to the specified key (`child` in this example) in the context of the parent work chain. In the next outline step of the parent work chain, the developer will then be able to access the finished child work chain through the context member `self.ctx.child`.

One might want to submit multiple subprocesses from within a single outline step, in the case where the subprocesses are independent from one another and can be executed in parallel. Both the `ToContext` container and the `to_context` method do not limit the number of subprocesses that can be assigned, as long as the keys to which they are assigned are unique. To prevent a developer from having to generate keys dynamically when one prefers to deal with an ordered list of results, the `ToContext` class and `to_context` method both support the `append_` free function. Consider the following example:

Listing 13: The `append_` free function can be used to alter the behavior of the `to_context` and `ToContext` constructs to append the created awaitable to a list instead of assigning it to a specific key.

(*continued on next column*)

(*continued*)

```
1 def submit_multiple child_workchains(self):
2         for i in range(10):
3                 self.to_context(children = append_(self.submit
                        (ChildWorkChain, **inputs))
```

The `append_` function in the `to_context` call ensures that the subprocesses, when finished, will be appended to a list in the context under the `children` key. In the next outline step, the developer can then access the list of process nodes that represent the completed subprocesses and iterate over them as with any other Python list.

2.2.7. Recording outputs

To emit outputs from a work chain, the class implements the `out` method, which takes two arguments, a string (used as the outgoing link label) and a node instance. At the point of calling, the `out` method merely records the new output node in memory and only at the end of the outline step will the engine commit the change to the database. It is at that point that the emitted output value is validated with respect to the output port as defined in the process specification of the work chain. When the execution of the work chain terminates, the emitted outputs are validated once more against the specification and if, for example, any required outputs have not been emitted, the work chain is marked as failed.

2.2.8. Aborting

At any point during the execution of a work chain, a developer might want to exit from the outline logic and terminate the execution prematurely. The engine can be instructed to terminate the execution of the work chain from within an outline step at any time, simply by returning a non-zero positive integer from the method, as shown in listing 14. The non-zero positive integer return value of the outline method is interpreted as an exit status, which is set on the node that represents the work chain in the provenance graph, and the process is terminated.

Listing 14: By returning a non-zero positive integer from any outline method, the engine is instructed to terminate the execution of the work chain and the return value is set as the `exit_status` attribute on the corresponding node.

```
1 def abort_from_this_step(self):
2         self.report('work chain will be terminated')
3         return 404
```

Alternatively, to provide an accompanying message for the reason of the exit, an instance of the `ExitCode` named tuple can be returned as well, which has the same effect as the integer exit status. The named tuple consists of an integer exit status and a string exit message. The tuple can be constructed manually, or it can be retrieved through the `exit_codes` attribute of the work chain, which is a container of the exit codes defined through the process specification of the work chain, as shown in listing 4.

Listing 15: By returning an `ExitCode` named tuple instance, the engine is instructed to terminate the execution of the work chain and the `exit_status` and `exit_message` of the return value is set on the corresponding node.

```
1   class AbortingWorkChain(WorkChain):
2
3           @classmethod
4           def define(cls, spec):
5                   super().define(spec)
6                   spec.exit_code(404, INEVITABLE_ERROR, 'this
    was unavoidable')
7                   spec.outline(cls.abort_straightaway)
8
9           def abort_straightaway(self):
10                  self.report('work chain will be terminated')
11                  return self.exit_codes.INEVITABLE_ERROR
```

The `self.exit_codes.INEVITABLE_ERROR` call retrieves the exit code instance that was defined in the process specification (line 6) and, when returned from the outline step, triggers the engine to terminate the

work chain. The exit status and message of the exit code are set on the corresponding attributes of the work chain node. Any potential caller of the work chain can then inspect these attributes and, based on their value, decide how to proceed.

2.2.9. Exposing of ports

As mentioned in the introduction, one of the major design goals of the workflow environment in AiiDA is to limit developers as little as possible in their freedom to design solutions, while providing them with the tools to write modular workflows. Modular workflows in this sense can be defined as workflows that perform a single well-defined task. Higher level workflows can then easily be built by wrapping these lower-level blocks. When a work chain wraps another work chain, it needs to 'expose' its input (and potentially output) ports, such that the caller of the top level work chain can pass in the required inputs. To make the process of wrapping a work chain within another workchain as simple as possible, and to prevent a developer from having to copy the port specification of the wrapped work chain manually, AiiDA implements the concept of automatic port exposing. To illustrate the concept of port exposing, consider the simple example of a `ParentWorkChain` wrapping a `ChildWorkChain`.

Listing 16: The `expose_inputs` method of the `ProcessSpec` class allows a work chain to automatically copy the ports of the work chain it is wrapping.

```
class ParentWorkChain(WorkChain):

  @classmethod
  def define(cls, spec):
    super().define(spec)
    spec.expose_inputs(ChildWorkChain)
    spec.outline(cls.run_child)

  def run_child(self):
    child_inputs = self.exposed_inputs(ChildWorkChain)
    child = self.submit(ChildWorkChain, **child_inputs)
    return ToContext(child = child)

class ChildWorkChain(WorkChain):

  @classmethod
  def define(cls, spec):
    super().define(spec)
    spec.input('a', valid_type = Int)
    spec.outline(cls.run_step)

  def run_step(self):
    self.report('running the ChildWorkChain')
```

The `expose_inputs` method by default copies over all the ports of the child work chain. Optionally, ports can be omitted through the `exclude` keyword, or specific ports can be selected with the `include` keyword. The work chain ports can also be exposed in a particular namespace by using the `namespace` keyword. This is especially useful if the exposed ports would otherwise overlap with existing ports with the same name.

2.2.10. Calculation jobs

High-performance computing resources rarely allow their users to directly run calculations on the system, but instead require resources to be requested from, or jobs to be submitted to, a scheduler, such as SLURM or PBS. Submitting calculations as jobs to these schedulers on remote computing resources is one of the most common activities in the workflow of a computational scientist, but involves a substantial amount of repetitive work. The job script has to be prepared and uploaded to the remote machine, including any other required input files for the calculation that is to be run. Subsequently, the job has to be submitted to the scheduler which will put it in a queue. The user must then monitor the queue to determine when the job is completed and the output files can be retrieved from the remote computing resource, to be optionally parsed and passed through post-processing tools.

This entire process is automated by AiiDA and implemented by the `CalcJob` class. A detailed description of how the `CalcJob` can be implemented for an arbitrary code that can be run on a remote cluster is beyond the scope of this paper and can be found in the extensive online documentation (aiida-core.readthedocs.io). Here, rather, we focus only on how the required steps of running a calculation job are realized automatically by AiiDA's engine.

In addition to having to run a calculation through a scheduler, another complexity of running calculations on high-performance computing resources is that these machines have to be remotely accessed, typically over an SSH connection. Since AiiDA is not required to run on the computing resource itself, it needs the ability to open a connection to perform the various operations involved with running calculation jobs. Any operation that requires opening a connection to the remote computing resource is referred to as a 'transport task', as it requires the SSH connection to "transport" the command from the local machine where AiiDA is running to the remote machine.

The life cycle of each calculation job knows four transport tasks that are executed in succession as shown in Fig. 3. For the first task, 'upload', the engine creates a new folder in the scratch space on the remote machine, into which the input files and job script are uploaded. Subsequently, the 'submit' task executes the command to submit the newly uploaded job script to the scheduler. If the job is successfully submitted, the output of the command is parsed to retrieve the unique identifier that the scheduler assigned to the job. The engine then uses this identifier to query the status of the job calculation in the 'update' task. Once the scheduler status indicates that the job has been completed, the engine invokes the 'retrieve' task, which retrieves all the files specified by the calculation plugin from the remote working directory to a local folder. This folder is attached as an output to the calculation node that represents the execution of the calculation job in the provenance graph. Finally, the engine parses the retrieved data and attaches the resulting output nodes to the formerly mentioned calculation node. However, since the data is now located on the local machine, this operation does not require an open SSH connection and is therefore not a transport task.

2.2.11. Error handling and robustness

As detailed in the previous section, the majority of operations required for running a calculation job to completion on a remote computing resource require the opening of an SSH connection and execution of a command over that connection. A variety of problems may occur during these steps. For example, the remote machine may be unreachable because the client machine lost its network connection, or the remote machine itself has network issues. Even when a connection is successfully established, there are still countless reasons why a remote operation may fail. The operation can be interrupted, timeout or simply fail on the remote machine. The latter can occur often for the transport tasks that interact with the scheduler, such as the 'submit' and 'update' tasks, when the scheduler is unreachable or overloaded.

In any case, problems that occur during transport tasks should not cause a failure of the running process but rather be dealt with elegantly.

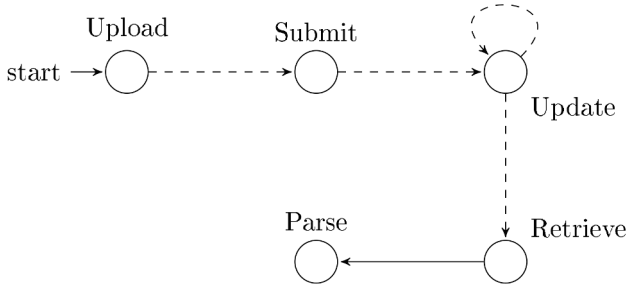

Fig. 3. Substates of the calculation job. A remote job is executed by transitioning through these states with each dashed transition occurring only when an open transport is available (prior to which the process is in a waiting state which is not shown).

More importantly, these problems are often temporary and are likely to resolve themselves or can be easily fixed by the user. Transport tasks therefore benefit enormously from an automated retry mechanism that detects problems and automatically retries the operation at a later point in time. This concept has been implemented in AiiDA's engine by an 'exponential-back-off-retry' mechanism, that works as follows. Each transport task is wrapped in an exponential-back-off-retry coroutine. This wrapper catches any exceptions that occur during the execution of the transport task, in which case it reschedules the same operation for execution at a later point in time. The wrapper reschedules failed tasks a maximum number of times, each time doubling the initial waiting interval, after which the parent process to which the transport task belongs, is paused. The initial waiting interval and the maximum number of retries are configurable per type of transport task.

In our experience, many problems are automatically resolved by this exponential-back-off-retry mechanism, but if this is not the case, the engine pauses the process instead of letting it except, giving the user the opportunity to investigate the problem. If the cause for the exception was external to the process itself and can be fixed, the paused process can be successfully resumed by the user. Alternatively, if the process itself was the source of the exception (e.g., a coding error in a parser or in a workflow step), the user can choose to kill it.

This mechanism of making calculation job processes robust is indispensable in high-throughput workflows. Without it, if, for instance, the network connection is lost when running a large number of complex and nested workflows in parallel, then all of the calculation jobs would except and, in a big cascade, all their parent processes too, resulting in a substantial loss of work. The exponential-back-off-retry mechanism combines a high degree of automation, by autonomously retrying failed tasks, and ultimately pausing tasks with problems instead of letting them except.

2.2.12. Transport queue

With the engine being primarily designed to operate under high-throughput conditions, one often runs many calculation jobs on a given remote resource, which requires many connections to be opened to that machine. However, remote computing resources typically limit the amount of connections that are allowed to be opened in a given time interval by a single client. Exceeding said limit can lead the client to be banned entirely from accessing the machine. To reduce the number of opened connections, while maintaining high-throughput capability, the engine bundles all connections through a 'transport queue'. Each worker (an independent Python process executing workflows - see section 3.1.1 for details) maintains one transport queue and the calculation jobs that it manages make requests for an open transport, instead of opening one themselves whenever they need it. The worker collects these requests and, at given a point in time, opens a single connection to the remote machine and distributes the transport to the processes that requested it. The transport queue guarantees that it opens a connection only once per safe-interval, a configurable minimum amount of time allowed between connections. This mechanism ensures that the maximum connection opening rate is never exceeded, even when running many concurrent calculation jobs on the same machine. The only limitation of this mechanism is that each worker maintains its own transport queue and there is no communication between those queues. This means that the promise of the safe-interval is only guaranteed per worker. However, by knowing how many workers are active (a value that the user can decide) the interval can of course be configured such that, on average, the connection rate is respected across all active workers.

2.2.13. Bundling scheduler update requests

The bundling of connections required by calculation jobs through the transport queue, as described in the previous paragraph, already relieves most of the load on remote computing resources when running in high-throughput mode. However, each active calculation job would still regularly perform the required remote operations, such as querying the scheduler for the state of the job, separately. When running in high-throughput mode, this scheme can still put unacceptable loads on the scheduler, despite the connection being shared. This, in turn, can render the scheduler unusable for all users of the computer cluster. Therefore, the AiiDA engine bundles all scheduler updates for calculation jobs (on each worker), very similarly to the transport queue for connections. When a calculation job needs to update its status, instead of polling the scheduler directly, it schedules an update request with the job manager of the worker. The job manager records these requests and, once a remote connection becomes available from the transport queue, issues a single scheduler update for the job identifiers that have registered themselves with it. The response is then parsed and the new status of each registered job is communicated to the corresponding calculation job. The combination of the bundling of connections and scheduler update requests ensures that the engine can run concurrent calculations jobs without overloading the remote computing resource.

3. Architecture

AiiDA's software architecture reflects several design goals that are informed, principally, by the needs of the high-throughput materials science community. These include the ability scale from running on laptops up to high-performance supercomputers, carrying out processes that range anywhere from fractions of a second to, potentially, weeks in execution time. Furthermore, it should be possible to have up to thousands of processes simultaneously active in a single instance. In terms of deployment configurations, AiiDA instances are typically installed on each user's workstation. However, there is the possibility to have a group or organisation-wide deployment or even a public-serving instance, as employed by the Materials Cloud ([32]).

As shown in Fig. 4, the workflow engine relies on two main external components for the execution of workflows:

a) The database engine (PostgreSQL [33]), used to persist the state of currently running processes, which doubles as a proxy to reflect the state of processes to the user, and,
b) the message broker (RabbitMQ [34]), which is responsible for delivering messages between the client(s) and the runner(s) (which may run in the same Python instance or even be on separate computers).

This decoupled approach has numerous advantages both in terms of flexibility of deployment configurations and for enabling a clear separation of concerns that makes it easier to write correct and robust code.

3.1. The engine

In order to meet a number of the design goals for the workflow

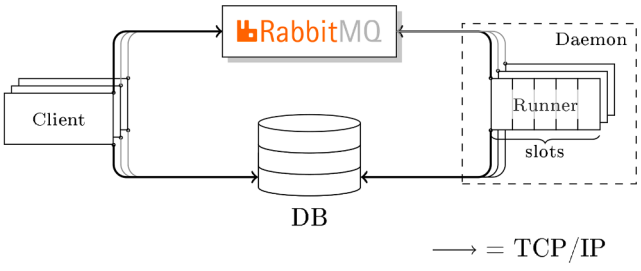

Fig. 4. Clients and runners maintain connections via TCP/IP to the database and RabbitMQ service enabling a rich set of possible configurations and corresponding usage scenarios. A daemon manages multiple, concurrent, runners each in a separate Python process (called a worker). Each runner executes serially but is nevertheless capable of handling multiple AiiDA processes by switching between slots when processes are waiting through the use of Python coroutines.

system relating to responsiveness, scaling and high-throughput capability, we rely heavily on events to trigger actions such as progressing a workflow from a state where it was waiting on something to complete, or to initiate the orderly termination of a running workflow. This is in stark contrast to polling based systems, where an entity that is waiting for a particular outcome has to periodically check if the event has occurred, often leading to unnecessary loads on the database and to poor responsiveness.

As with many event-based systems (e.g. graphical user interfaces, computer games) AiiDA uses an event loop to achieve this, which in our case is provided by Python's built-in `asyncio`. Furthermore, this enables multiple AiiDA processes to be managed by a single Python instance, despite the lack of multithreading support, as coroutines can be scheduled for execution and can yield to others while they are waiting for some action to complete. The use of coroutines mitigates another issue, which is that database servers typically have a low default connection limit (100 in the case of PostgreSQL) and in threaded environments it is not uncommon for each thread to have a separate connection. Lastly, writing correct multithreaded code is extremely complex and would be difficult, even for experienced programmers, particularly given that we place no restriction on the API calls that can be made.

The entire stack of Python components needed to execute workflows are brought together in the `Runner` class which provides the event loop, persistence, communication, transport (e.g., SSH) and other functionality, some of which are described in greater detail below. Thanks to the event loop, each runner can run any number of workflow processes concurrently (within memory limits). We call the number processes that can run on a single runner the number of process slots.

3.1.1. The daemon

The runner can be used in a local interpreter which is particularly useful for testing and debugging, or as a daemon worker which is simply a Python process exclusively executing a runner. However, in most production environments the user wishes to launch a daemon that can manage one or more workers, automatically restarting them if they happen to crash. In AiiDA, we use the Circus [35] library to achieve this. Circus provides the ability to start multiple operating system processes, automatically restarting them if they crash. In addition, it can show information about their current resource usage and dynamically increase the number of workers in the pool.

With the daemon one can scale both vertically (multiple slots per runner) and horizontally (multiple workers each with one runner) as shown in Fig. 5. For workloads that require significant in-Python processing it is preferable to scale the number of workers while workloads involving many remote calculations can just as well scale the number of slots per runner, minimising the load on the computer running the daemon with little loss in throughput. The hard limit is reached when the number of runners equals the maximum number of available database connections (however, this is user configurable if they have access to the database settings). In benchmarking we have shown [8] that on a modest workstation (Intel Xeon E5-2623 v3 CPU, 64 GB of RAM) using 12 workers it was possible to reach a throughput of 35,000 processes per hour for a trivial workflow including one job calculation that was being executed on the same machine as the daemon. Ignoring the overheads of the SSH protocol (from the machine to itself) and of running the simple job calculation (a bash script that adds two numbers) this gives a rough indication of the overheads of the AiiDA engine including all the database operations involved in storing the provenance. Clearly, the performance of the AiiDA engine far exceeds the throughput of any real-world HPC workflow.

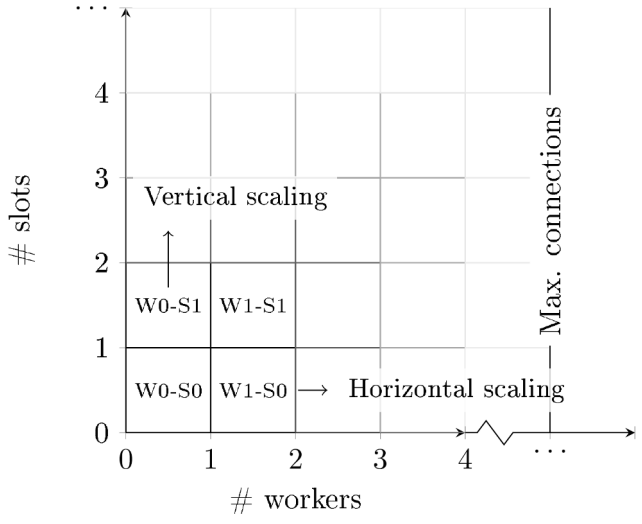

Fig. 5. Scaling of the number of workers and slots per runner. The total number of concurrent AiiDA processes possible is the product of the values on the two axes.

3.2. The process

The principal object of AiiDA's workflow engine is the `Process` class. All specific classes (`WorkChain`s, `CalcJob`s, etc.) derive from `Process` (or a subclass thereof) and in so doing inherit a large swathe of common functionality and features. The `Process` class itself is modeled as an extended state machine, meaning that it is composed of a finite-state machine, shown in Fig. 6, where each state can have internal data members as part of its extended state.

This is a pattern common in event-driven systems as it provides a consistent way to model the current state, as well as event hooks that can be used as triggers to perform actions either internal or external to the process during state transitions. The event hooks themselves come in the form of process member functions, e.g.:

Listing 17: AiiDA's `Process` state transition hooks. These are invaluable for being able to guarantee that certain actions are performed when a state transition occurs.

```
# Entering a new state
def on_entering(self, state):
        ...
# Just entered the new state from 'from_state'
def on_entered(self, from_state):
        ...
# About to exit the current state
def on_exiting(self):
        ...
```

By using these hooks it is possible to schedule actions that should always be executed at the various points of a state transition and therefore guarantee a consistent state once the `on_entered` hook has finished. One use of these hooks in AiiDA is to reflect the current state of the process back to the database including saving a checkpoint. The state transition hooks are also used to send broadcast messages that allow listeners (which can potentially be on remote machines) to be updated of

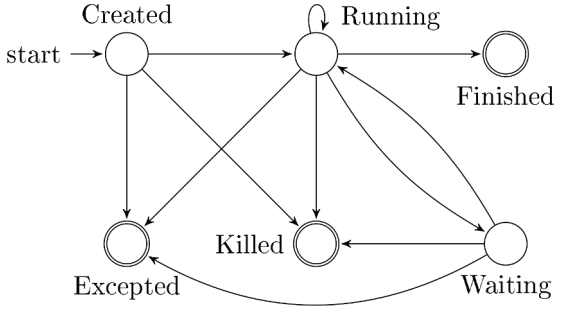

Fig. 6. The process state machine. Terminal states are represented with a double circle.

state changes as they happen (see section 3.3).

A typical process progresses through the various states, potentially running several member functions or waiting on other processes to finish. However, if an exception is propagated up to the `Process` level, it will be caught and the `Process` transitions to the terminal `EXCEPTED` state, whereupon a log entry is created containing, amongst other things, the Python stack trace. The other way to terminate a `Process` prematurely is to call the kill method.

3.2.1. Persistence

A key requirement for the engine is that the progress of processes is check-pointed and persisted such that, in the case of an orderly or disorderly shutdown, the AiiDA engine can continue from the last clean state upon being restarted. In order to achieve this we use checkpoints that are written to the database at process state transitions. Specifically, the context of the process, outputs, and some metadata are saved to a dictionary which is then serialized to the database by an AiiDA specific persister as shown in Fig. 7.

3.3. Communication

The workflow engine relies heavily on messaging for the external control of processes and to maintain high-throughput whilst ensuring fault tolerance. This is achieved by using a "message broker", in our case RabbitMQ. Message brokers typically take responsibility for guaranteeing the durability and atomicity of messages, allowing the application to focus on the business logic. In RabbitMQ's case the user installs a service that client software interacts with via a TCP port. The routing and persistence of messages are handled internally by RabbitMQ.

To facilitate AiiDA's interaction with RabbitMQ, we developed the kiwiPy ([36]) library. KiwiPy significantly simplifies the process of interacting with RabbitMQ and provides the ability to offload communication to a separate thread. This is essential for AiiDA as described below in subsection 'task queues'. In addition to task queues kiwiPy provides AiiDA the ability to send Remote Procedure Call (RPC) and broadcast messages.

3.3.1. Task queues

These are used to schedule new processes to be run. There is a major advantage to using RabbitMQ in that it provides certain guarantees about messages, depending on the chosen settings. For our task queues AiiDA uses persistent messages, which are persisted to disk such that they survive intentional or unintentional restarts of the machine. As such, a job is guaranteed to never be lost once delivered from the client to the broker. Furthermore, RabbitMQ expects acknowledgements for tasks that have been completed. If it loses connection with the runner, it automatically requeues the task again, until completion is acknowledged. This mechanism relies on the use of periodic messages, called heartbeats, to which the runner must respond in a timely manner, otherwise, upon missing two consecutive responses, RabbitMQ assumes the runner to be dead and triggers the rescheduling mechanism. It is for this reason that kiwiPy runs a separate thread so that, even when AiiDA processes are under a heavy and blocking workload, it is able to respond to heartbeats.

3.3.2. RPC

As suggested by the name, these kinds of messages are used to invoke a procedure (in our case a function or method) on the receiving process and deliver the result of the process back to the caller. This is used primarily to pause, play and kill active processes.

3.3.3. Broadcast

This involves sending a single message to any registered listeners with no possibility for them to send a direct response. These are used for two purposes: to pause, play or kill groups of processes and to control the flow between them. Parent processes that have spawned children can choose to wait for a child to complete before continuing their execution. This is facilitated by registering itself as a listener to broadcasts from the child and yielding until it receives the child "terminated" message. This mechanism is what enables the functionality of the `to_context` work chain construct, as described in paragraph 2.2.6, notifying the work chain that it can continue as the process it was waiting for has completed.

4. Conclusions

In this article we have described the decisions that have guided the development and shaped the internals of AiiDA's workflow engine, in the recognition that often the insights gained in the development process can be as valuable as the finished product itself. AiiDA has a fairly broad set of challenging goals and target use cases. To meet these, we have incorporated a number of advanced programming techniques such as coroutines, extended finite-state machines, event-driven programming, futures, and a number of technologies commonly employed in industry, but less common in academia, such as RabbitMQ and PostgreSQL. Being mindful of the inherent complexity of this system, and the tasks it aims to address, we have attempted to create a user-friendly API that allows non-experts to write powerful, modular, robust and auto-documenting workflows with full provenance automatically stored as they run. With the integration of AiiDA's plugin system, sharing these workflows with the public is made simple, enabling the wider community to reuse the scientific knowledge encoded within. While the engine is particularly well suited to manage high-throughput workflows whose steps involve simulations running on high-performance computing infrastructures, it is versatile enough to also effortlessly run code on smaller machines such

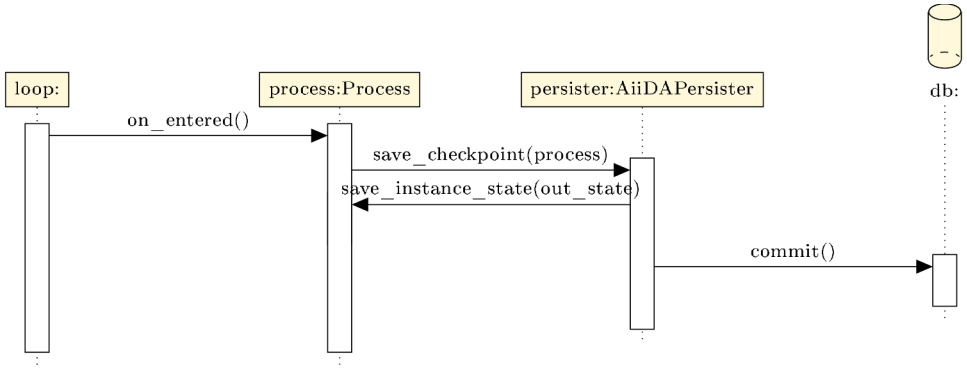

Fig. 7. A UML activity diagram showing the persistence of the internal state of a process at a state transition. The vertical bars show the period during witch the entity (at the top) is active and therefore defines its scope. When the persister gets a request to save the checkpoint is calls the Process and requests that it populates a dictionary, `out_state` which is then serialized and committed to the database.

as personal desktops.

With the release of AiiDA 1.0, its extensible and modular nature has accelerated its adoption within the materials discovery community with over 100 supported simulation code executables and many workflows [30] available at the time of writing, many of which have contributed directly to published scientific works. These are all encouraging signs, as the ultimate goal of AiiDA is to provide the community with a useful tool that can be used as part of an interoperable, FAIR, computational infrastructure to accelerate scientific discovery. As the scientific community transitions to the exascale era, there is little doubt that such tools will have a greater and greater role to play in the daily activities of researchers.

Data availability

No new datasets were generated or analysed during this study.

CRediT authorship contribution statement

Martin Uhrin: Conceptualization, Software, Writing - original draft. **Sebastiaan P. Huber:** Conceptualization, Software, Writing - original draft. **Jusong Yu:** Software, Writing - review & editing. **Nicola Marzari:** Conceptualization, Writing - review & editing, Supervision, Project administration, Funding acquisition. **Giovanni Pizzi:** Conceptualization, Software, Writing - review & editing, Visualization, Supervision, Project administration, Funding acquisition.

Declaration of Competing 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.

Acknowledgments

The authors thank Dominik Gresch, Tiziano Müller and Jens Bröder for detailed technical discussions. We thank the whole AiiDA team for contributions throughout the development process.

MU would like to thank Matthieu Mottet for valuable discussions regarding asyncio and coroutines in Python.

We thanks the Swiss Data Science Center for a stimulating exchange regarding data provenance frameworks.

We thank Sonia Collaud for help in creating the graphical abstract.

This work is supported by the MARVEL National Centre for Competency in Research funded by the Swiss National Science Foundation (grant agreement ID 51NF40-182892), the European Centre of Excellence MaX “Materials design at the Exascale” (Grant No. 824143), by the Swiss Platform for Advanced Scientific Computing (PASC) and by the swissuniversities P-5 “Materials Cloud” project (grant agreement ID 182-008). This work was supported by grants from the Swiss National Supercomputing Centre (CSCS) under project ID s836. We acknowledge PRACE for awarding us access to Piz Daint at CSCS, Switzerland under Grant No. 2016153543.

References

[1] D. Talia, ISRN Software Eng. 2013 (2013) 1.
[2] M.D. Wilkinson, M. Dumontier, I.J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L.B. da Silva Santos, P. E. Bourne, J. Bouwman, A. J. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C.T. Evelo, R. Finkers, A. Gonzalez-Beltran, A.J. Gray, P. Groth, C. Goble, J.S. Grethe, J. Heringa, P.A. ’t Hoen, R. Hooft, T. Kuhn, R. Kok, J. Kok, S.J. Lusher, M. E. Martone, A. Mons, A. L. Packer, B. Persson, P. Rocca-Serra, M. Roos, R. van Schaik, S.-A. Sansone, E. Schultes, T. Sengstag, T. Slater, G. Strawn, M. A. Swertz, M. Thompson, J. van der Lei, E. van Mulligen, J. Velterop, A. Waagmeester, P. Wittenburg, K. Wolstencroft, J. Zhao, B. Mons, Scientific Data 3 (2016), 10.1038/sdata.2016.18.
[3] J.P.A. Ioannidis, D.B. Allison, C.A. Ball, I. Coulibaly, X. Cui, A.C. Culhane, M. Falchi, C. Furlanello, L. Game, G. Jurman, J. Mangion, T. Mehta, M. Nitzberg, G. P. Page, E. Petretto, V. van Noort, Nat. Genet. 41 (2009) 149.
[4] R.D. Peng, Science 334 (2011) 1226.
[5] C. Stoddart, Nature (2016), https://doi.org/10.1038/d41586-019-00067-3.
[6] D.B. Allison, A.W. Brown, B.J. George, K.A. Kaiser, Nature 530 (2016) 27.
[7] C. Goble, S. Cohen-Boulakia, S. Soiland-Reyes, D. Garijo, Y. Gil, M.R. Crusoe, K. Peters, D. Schober, Data Intell. 2 (2020) 108.
[8] 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, G. Pizzi, Sci. Data 7 300 (2020) 2003.12476.
[9] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, S. Mock, in: Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004, IEEE.
[10] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M.R. Pocock, A. Wipat, P. Li, Bioinformatics 20 (2004) 3045.
[11] I. Taylor, M. Shields, I. Wang, O. Rana, J. Grid Comput. 1 (2003) 199.
[12] G. von Laszewski, M. Hategan, D. Kodeboyina, in: Workflows for e-Science, Springer London, 2007, pp. 340–356.
[13] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, M. Wieczorek, in: The 6th IEEE/ACM International Workshop on Grid Computing, 2005, IEEE, 2005.
[14] T. Fahringer, J. Qin, S. Hainzer, in: CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005, IEEE, 2005.
[15] B. Chapman, J. Chilton, M. Heuer, A. Kartashov, D. Leehr, H. Ménager, M. Nedeljkovich, M. Scales, S. Soiland-Reyes, L. Stojanovic, Common Workflow Language, v1.0 (figshare, 2016).
[16] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A. Laity, J.C. Jacob, D.S. Katz, Sci. Programm. 13 (2005) 219.
[17] C.S. Adorf, P.M. Dodd, V. Ramasubramani, S.C. Glotzer, Comput. Mater. Sci. 146 (2018) 220.
[18] Y. Babuji, I. Foster, M. Wilde, K. Chard, A. Woodard, Z. Li, D.S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J.M. Wozniak, in Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing – HPDC 2019, ACM Press, 2019.
[19] A. Jain, S.P. Ong, W. Chen, B. Medasani, X. Qu, M. Kocher, M. Brafman, G. Petretto, G.-M. Rignanese, G. Hautier, D. Gunter, K.A. Persson, Concurr. Comput. Pract. Exp. 27 (2015) 5037.
[20] The JavaScript Object Notation (JSON) Data Interchange Format, Tech. Rep., 2014.
[21] S. Curtarolo, W. Setyawan, G.L. Hart, M. Jahnatek, R.V. Chepulskii, R.H. Taylor, S. Wang, J. Xue, K. Yang, O. Levy, M.J. Mehl, H.T. Stokes, D.O. Demchenko, D. Morgan, Comput. Mater. Sci. 58 (2012) 218.
[22] K. Mathew, J.H. Montoya, A. Faghaninia, S. Dwarakanath, M. Aykol, H. Tang, I. heng Chu, T. Smidt, B. Bocklund, M. Horton, J. Dagdelen, B. Wood, Z.-K. Liu, J. Neaton, S.P. Ong, K. Persson, A. Jain, Comput. Mater. Sci. 139 (2017) 140.
[23] T. Mayeshiba, H. Wu, T. Angsten, A. Kaczmarowski, Z. Song, G. Jenness, W. Xie, D. Morgan, Comput. Mater. Sci. 126 (2017) 90.
[24] J.E. Saal, S. Kirklin, M. Aykol, B. Meredig, C. Wolverton, JOM 65 (2013) 1501.
[25] K. Lejaeghere, G. Bihlmayer, T. Bjorkman, P. Blaha, S. Blugel, V. Blum, D. Caliste, I. E. Castelli, S.J. Clark, A.D. Corso, S. de Gironcoli, T. Deutsch, J.K. Dewhurst, I.D. Marco, C. Draxl, M.D. ak, O. Eriksson, J.A. Flores-Livas, K.F. Garrity, L. Genovese, P. Giannozzi, M. Giantomassi, S. Goedecker, X. Gonze, O. Granas, E.K.U. Gross, A. Gulans, F. Gygi, D. R. Hamann, P. J. Hasnip, N. A. W. Holzwarth, D. I. an, D. B. Jochym, F. Jollet, D. Jones, G. Kresse, K. Koepernik, E. Kucukbenli, Y.O. Kvashnin, I.L.M. Locht, S. Lubeck, M. Marsman, N. Marzari, U. Nitzsche, L. Nordstrom, T. Ozaki, L. Paulatto, C.J. Pickard, W. Poelmans, M.I.J. Probert, K. Refson, M. Richter, G.-M. Rignanese, S. Saha, M. Scheffler, M. Schlipf, K. Schwarz, S. Sharma, F. Tavazza, P. Thunstrom, A. Tkatchenko, M. Torrent, D. Vanderbilt, M.J. van Setten, V.V. Speybroeck, J.M. Wills, J.R. Yates, G.-X. Zhang, S. Cottenier, Science 351 (2016) aad3000.
[26] http://www.pbsworks.com/Product.aspx?id=1.
[27] https://computing.llnl.gov/linux/slurm/.
[28] https://www.oracle.com/technetwork/oem/grid-engine-166852.html.
[29] http://www.adaptivecomputing.com/products/open-source/torque/.
[30] https://aiidateam.github.io/aiida-registry/.
[31] https://pip.pypa.io/en/stable/.
[32] L. Talirz, S. Kumbhar, E. Passaro, A.V. Yakutovich, V. Granata, F. Gargiulo, M. Borelli, M. Uhrin, S.P. Huber, S. Zoupanos, C.S. Adorf, C.W. Andersen, O. Schütt, C. A. Pignedoli, D. Passerone, J. VandeVondele, T.C. Schulthess, B. Smit, G. Pizzi, N. Marzari, Sci. Data 299 (2020) 7, 2003.12510.
[33] http://www.postgresql.org/.
[34] https://www.rabbitmq.com.
[35] https://circus.readthedocs.io/.
[36] M. Uhrin, S. Huber, J. Open Source Software 5 (2020) 2351.